Gestion autonomique d'applications dynamiques sûres et résilientes

Thomas Calmant

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Thomas Calmant

Thèse dirigée par **M. Didier Donsez**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
et de **L'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Gestion Autonomique d'Applications Dynamiques Sûres et Résilientes

Thèse soutenue publiquement le **19 Octobre 2015**,
devant le jury composé de :

M. Vivien Quéma
Professeur à Grenoble INP, Président
M. Lionel Seinturier
Professeur à l'Université de Lille 1, Rapporteur
M. Gaël Thomas
Professeur à Télécom SudParis, Rapporteur
Mme. Sonia Ben Mokhtar
Chargée de Recherche au CNRS, LIRIS Lyon, Examinatrice
M. Walter Rudametkin
Maître de Conférences à l'Université de Lille 1, Examinateur
M. Didier Donsez
Professeur à l'Université Joseph Fourier, Grenoble 1, Directeur de thèse
M. Olivier Gattaz
Architecte Logiciel chez isandlaTech, Co-Encadrant de thèse

REMERCIEMENTS

« Ce n'est pas tant l'aide de nos amis qui nous aide que notre confiance dans cette aide. »

ÉPICURE

Mes premiers remerciements sont adressés à Didier Donsez et Olivier Gattaz, pour m'avoir proposé l'aventure qu'a été cette thèse, pour leur encadrement et pour leur soutien tout au long de ces trois années.

Je tiens à remercier très chaleureusement Vivien Quéma pour l'honneur qu'il m'a fait de présider ce jury mais aussi pour nos débats œnologiques tout au long de mon séjour au laboratoire.

Je souhaite exprimer ma profonde reconnaissance à Lionel Seinturier et Gaël Thomas qui ont accepté d'être les rapporteurs de cette thèse et qui ont apporté leur jugement constructif. Ma reconnaissance va également à Sonia Ben Mokthar pour avoir accepté de participer au jury de thèse et de porter son intérêt sur mon travail.

Je n'oublie pas de remercier Walter Rudametkin, certes pour avoir accepté de faire partie du jury, mais surtout pour m'avoir initié au travail de recherche, m'avoir conseillé de réaliser cette thèse et enfin m'avoir poussé à aller jusqu'au bout de mes travaux.

Je remercie également tous les membres de l'équipe ERODS, pour leur accueil, leurs conseils et toutes nos discussions autour d'un café. Je remercie particulièrement Noël pour m'avoir appris à prendre du recul sur mon travail mais aussi pour toutes nos fameuses dégustations.

Ce manuscrit n'aurait pas pu voir le jour sans le soutien inconditionnel de mes amis, Dijonnais et Grenoblois. Julien, Guillaume, Cédric, Virginie, Christophe, Nicolas, Cindy, François-Karim, Bassem, Ahmad, Shadi, Asan et ceux qui ont partagé mes doutes et mes joies : merci à vous !

Je remercie également ceux de l'autre côté du bar pour toutes les discussions qui m'ont remonté le moral : Antho, Flo, Lisa, Wendy, Manu, Jean-Mi, Julien, Frenzi, Ludo, Claire, Caro, et tant d'autres :)

Enfin, je remercie toute ma famille pour le soutien qu'elle m'a apporté tout au long de cette thèse, et tout particulièrement mes parents, qui m'ont toujours motivé et permis de poursuivre mes études jusqu'à cet aboutissement.

Un grand merci à tous !

RÉSUMÉ

Les architectures orientées services (SOA) sont considérées comme le moyen le plus avancé pour réaliser et intégrer rapidement des applications modulaires et flexibles. Dans ce domaine, les plates-formes SOA à disposition des développeurs et des architectes de produits logiciels sont multiples ; les deux plus évoluées d'entre elles étant SCA [1] et OSGi [2]. Une application s'appuyant sur l'une de ces plates-formes peut ainsi être assemblée avec le minimum de composants nécessaires à la réalisation de ses tâches, afin de réduire sa consommation de ressources et d'augmenter sa maintenabilité. De plus, ces plates-formes autorisent l'ajout de composants greffons qui n'étaient pas connus lors des phases initiales de la réalisation du produit. Elles permettent ainsi de mettre à jour, d'étendre et d'adapter continuellement les fonctionnalités du produit de base ou des services techniques nécessaires à sa mise en production, sans interruption de service. Ces capacités sont notamment utilisées dans le cadre du paradigme *DevOps* et, plus généralement, pour mettre en œuvre le déploiement continu d'artefacts.

Cependant, l'extensibilité offerte par ces plates-formes peut diminuer la fiabilité globale du système : une tendance forte pour développer un produit est l'assemblage de composants provenant de tierces-parties. De tels composants peuvent être d'une qualité inconnue voire douteuse. En cas d'erreur, de détérioration des performances, etc., il est difficile de diagnostiquer les composants ou combinaisons de composants incriminés. Il devient indispensable pour le producteur d'un logiciel de déterminer la responsabilité des différents composants impliqués dans un dysfonctionnement.

Cette thèse a pour objectif de fournir une plate-forme, Cohorte, permettant de concevoir et d'exécuter des produits logiciels extensibles et résilients aux dysfonctionnements d'extensions non qualifiées. Les composants de tels produits pourront être développés dans différents langages de programmation et être déployés (ajout, mise à jour et retrait) en continu et sans interruption de service. Notre proposition adopte pour principe d'isoler les composants considérés comme instables ou peu sûrs. Le choix des composants à isoler peut être décidé par l'équipe de développement et l'équipe opérationnelle, à partir de leur expertise, ou bien déterminé à partir d'une combinaison d'indicateurs. Ces derniers évoluent au cours du temps pour refléter la fiabilité des composants. Par exemple, des composants peuvent être considérés fiables après une période de quarantaine ; une mise à jour peut entraîner la dégradation de leur stabilité, etc.. Par conséquent, il est indispensable de remettre en cause les choix initiaux dans l'isolation des composants afin, dans le premier cas, de limiter le coup des communications entre composants et, dans le deuxième cas, de maintenir le niveau de fiabilité du noyau critique du produit.

1. `http://www.oasis-opencsa.org/sca`
2. `http://www.osgi.org`

ABSTRACT

Service-Oriented architectures (SOA) are considered the most advanced way to develop and integrate modular and flexible applications. There are many SOA platforms available for software developers and architects ; the most evolved of them being SCA [1] and OSGi [2]. An application based on one of these platforms can be assembled with only the components required for the execution of its tasks, which helps decreasing its resource consumption and increasing its maintainability. Furthermore, those platforms allow adding plug-ins at runtime, even if they were not known during the early stages of the development of the application. Thus, they allow updating, extending and adapting the features of the base product or of the technical services required for its execution, continuously and without outage. Those capabilities are applied in the *DevOps* paradigm and, more generally, to implement the continuous deployment of artifacts.

However, the extensibility provided by those platforms can decrease the overall reliability of the system : a strong tendency in software development is the assembly of third-parties components. Such components may be of unknown or even questionable quality. In case of error, deterioration of performance, ... it is difficult to identify the implicated components or combinations of components. It becomes essential for the software producer to determine the responsibility of the various components involved in a malfunction.

This thesis aims to provide a platform, Cohorte, to design and implement scalable software products, resilient to malfunctions of unqualified extensions. The components of such products may be developed in various programming languages and be deployed continuously (adding, updating and withdrawal) and without interruption of service. Our proposal adopts the principle of isolating the components considered unstable or insecure. The choice of the components to be isolated may be decided by the development team and by the operational team, from their expertise, or determined from a combination of indicators. The latters evolve over time to reflect the reliability of components. For example, components can be considered reliable after a quarantine period ; an update may result in deterioration of stability, ... Therefore, it is essential to question the initial choices in isolating components to limit, in the first case, the scope of communications between components and, in the second case, to maintain the reliability of the critical core of the product.

1. `http://www.oasis-opencsa.org/sca`
2. `http://www.osgi.org`

TABLE DES MATIÈRES

Remerciements **iii**

Résumé **v**

Abstract **vii**

1 Introduction **1**
1.1 Contexte 1
1.2 Problématique 4
1.3 Objectifs 5
1.4 Contributions 6
1.5 Organisation du manuscrit 7

2 État de l'art **9**
2.1 Introduction 10
2.2 Approche Orientée Composant 10
2.2.1 Définition 10
2.2.2 Implémentations 11
2.3 Approche Orientée Service 13
2.3.1 Définition 13
2.3.2 Implémentations 15
2.4 Approche à Composant Orienté Service 18
2.4.1 Définition 18
2.4.2 Implémentations 18
2.5 Distribution de services 20
2.5.1 Eclipse Communications Framework 21
2.5.2 ETHZ R-OSGi 21
2.5.3 Apache CXF DOSGi 22
2.5.4 OW2 RoSe et Fuchsia 22
2.5.5 Résumé 23

2.6 Isolation de composants 23
2.6.1 Isolation par processus 24
2.6.2 Isolation par conteneur 27
2.6.3 Isolation par machine virtuelle système 28
2.7 Canevas de distribution de composants 33
2.7.1 SCA 33
2.7.2 HighFrame 36
2.7.3 AIOLOS 38
2.8 Conclusion 39

3 Contribution 1 : iPOPO **41**
3.1 Introduction 42
3.2 Pelix : Architecture Orientée Service 43
3.2.1 Unité de déploiement 43
3.2.2 Services 45
3.3 iPOPO : Composants Orientés Service 46
3.3.1 Métamodèle 46
3.3.2 Fabriques de composants 47
3.3.3 Conteneur et handlers 49
3.3.4 Cycle de vie des instances 50
3.3.5 Liaisons inter-composants 52
3.4 Services distants 52
3.4.1 Architecture 53
3.4.2 Protocoles supportés 55
3.5 Services utilitaires 55
3.5.1 Pelix Shell 56
3.5.2 Service HTTP 57
3.5.3 Event Admin 57
3.5.4 Configuration Admin 58
3.6 Comparaison avec les modèles existants 59
3.6.1 Cycle de développement d'un composant 59
3.6.2 Liaisons entre composants 60
3.6.3 Propriétés extra-fonctionnelles 63
3.7 Conclusion 64

4 Contribution 2 : Cohorte Herald **67**
4.1 Introduction 67
4.2 Modèle 68

4.2.1 Concepts . . . 68
4.2.2 Format de message . . . 69
4.2.3 Structure . . . 70
4.3 Découverte de pairs . . . 71
4.4 Émission de messages . . . 72
4.4.1 Modes de transmission . . . 72
4.4.2 Implémentation . . . 74
4.5 Services distants . . . 74
4.5.1 Découverte de services . . . 75
4.5.2 Messages d'appels distants . . . 77
4.6 Shell distant . . . 78
4.7 Conclusion . . . 78

5 Contribution 3 : Cohorte Runtime 81
5.1 Introduction . . . 81
5.2 Modèle . . . 82
5.3 Fichier de configuration d'un isolat . . . 82
5.4 Architecture . . . 84
5.5 Cycle de vie d'un isolat . . . 86
5.5.1 Démarrage d'un isolat par *bootstrap* . . . 86
5.5.2 Démarrage d'un isolat *exe* . . . 88
5.5.3 Disparition d'un isolat . . . 88
5.5.4 Arrêt de la plate-forme . . . 89
5.6 Conclusion . . . 90

6 Contribution 4 : Cohorte Composer 93
6.1 Introduction . . . 93
6.2 Architecture . . . 94
6.3 Modèle de composition . . . 96
6.3.1 Composition . . . 96
6.3.2 Composant . . . 97
6.3.3 Comparatif . . . 98
6.4 Implémentation . . . 99
6.4.1 Surveillance et notation des composants . . . 99
6.4.2 Distribution des composants par vote . . . 100
6.4.3 Instanciation des composants . . . 103
6.5 Gestion autonomique de la composition . . . 105
6.5.1 Définition . . . 105

6.5.2 Comparaison avec Cohorte Composer 107
6.6 Conclusion 108

7 Expérimentations et validations **111**
7.1 Introduction 111
7.2 Agilium NG 112
7.3 Prédis MHI 114
7.4 Robot Nao et OpenHab 115
7.5 Évaluation des performances 116
7.5.1 iPOPO 116
7.5.2 Services Distants 120
7.5.3 Cohorte 126
7.6 Conclusion 129

8 Conclusion et perspectives **131**
8.1 Synthèse 131
8.1.1 Rappel de la problématique 131
8.1.2 Approche et exigences 132
8.1.3 Contributions 133
8.2 Perspectives 136
8.2.1 Travaux en cours 136
8.2.2 Travaux envisagés 138

Bibliographie **141**

TABLE DES FIGURES

1.1 Cycle de vie d'un projet selon la méthode DevOps 3

2.1 Représentation d'un composant 11
2.2 Interactions dans le modèle SOA 14
2.3 Les différentes échelles d'isolation 32
2.4 Composant SCA 33
2.5 Composite SCA 34
2.6 Modèle de composant Fraclet 36
2.7 Vue globale de la plate-forme AIOLOS 38

3.1 Cycle de vie d'un bundle Pelix 44
3.2 Métamodèle iPOPO 47
3.3 Cycle de vie d'un composant iPOPO 51
3.4 Architecture des *Pelix Remote Services* 53
3.5 Diagramme de séquence : export et import d'un nouveau service 54

4.1 Vision globale des composants de Herald 70
4.2 Synchronisation des annuaires Herald 72
4.3 Transmission de message Herald 73
4.4 Échange de services entre pairs 76

5.1 Vue globale de Cohorte Runtime 85

6.1 Vue globale de Cohorte Composer 94
6.2 Représentation d'une boucle autonomique 106

7.1 Vue globale d'Agilium NG 113
7.2 Représentation de la table 7.1 118
7.3 Temps d'instanciation des composants `B` et `C` pour 100 et 1 000 composants `A` . . 119

8.1 Tunnel Herald 137
8.2 Routage de messages Herald 138

LISTE DES TABLEAUX

3.1 Extrait de propriétés spécifiées par OSGi 46
3.2 Classification selon le cycle de vie des composants 59
3.3 Classification selon la construction - Spécification des interfaces 61
3.4 Classification selon la construction - Liaisons et interactions 62
3.5 Classification selon le traitement des Propriétés Extra-Fonctionnelles 63
3.6 Lignes de code d'iPOPO 64

4.1 Messages de découvertes 75
4.2 Lignes de code de Cohorte Herald 79

5.1 Lignes de code de Cohorte Runtime 91

6.1 Comparaison des descriptions de composants Cohorte, Fraclet et SCA 98
6.2 Niveaux autonomiques 107
6.3 Comparaison avec les canevas de distribution de composants 109
6.4 Lignes de code de Cohorte Composer 110

7.1 Temps d'instanciation de composants des fabriques tests 117
7.2 Temps d'instanciation de composants des fabriques `B` et `C` 118
7.3 Durées moyennes des appels distants à la méthode `dummy` 121
7.4 Durées moyennes des appels distants à la méthode `single_arg` 122
7.5 Taille en octets du corps des requêtes d'appel distant selon le type d'argument . . 123
7.6 Durées moyennes des appels distants à la méthode `var_args` 124
7.7 Durées moyennes par appel lors d'appels distants simultanés à la méthode `dummy` 125
7.8 Temps moyen de démarrage des isolats 126
7.9 Temps de démarrage d'une application en fonction du nombre de composants . . 127
7.10 Temps de redémarrage d'une composition en fonction du nombre de composants 128

8.1 Lignes de code des différentes contributions 136

CHAPITRE 1

INTRODUCTION

> *Begin at the beginning," the King said, gravely, "and go on till you come to an end ; then stop.*
>
> Lewis CARROLL, *Alice's Adventures in Wonderland*, 1865

Sommaire

1.1 Contexte **1**
1.2 Problématique **4**
1.3 Objectifs **5**
1.4 Contributions **6**
1.5 Organisation du manuscrit **7**

1.1 Contexte

Depuis les années quatre-vingt-dix, le Génie Logiciel est passé d'un modèle fortement planifié – emprunté à l'industrie lourde – aux modèles agiles [BBVB+01], dans lesquels les parties prenantes (*stakeholders*) interagissent plus efficacement entre elles. Les *utilisateurs*, *i.e.* ceux qui utiliseront le produit, les *clients*, *i.e.* ceux qui achètent le produit, les *développeurs*, *i.e.* ceux qui le conçoivent, et enfin le *fournisseur*, *i.e.* l'organisation qui gère le développement du produit, constituent les parties prenantes d'un projet. Le modèle industriel a été initialement préféré du fait de sa compatibilité avec le fonctionnement interne des grandes entreprises, l'avancement d'un projet se mesurant selon les documents produits à la fin de chacune de ses phases. En effet, dans ce modèle tel que décrit dans [Som11], un client et son fournisseur se mettent initialement d'accord sur les spécifications d'un produit, qui ne pourront être que très légèrement modifiées avant la livraison du produit final. Dans cette méthodologie de projet, des prototypes peuvent être livrés voire simplement montrés aux clients afin de valider leur respect des spécifications et de légèrement faire évoluer ces dernières si besoin est. Les cycles les plus fréquemment utilisés sont le *cycle en cascade*, dont les étapes du projet se déroulent successivement, avec la possibilité de rétroagir sur l'étape précédente ; le *cycle en V*, où le résultat de chaque étape de validation du produit développé est confronté à sa définition dans la spécification ; et le *cycle itératif*, définissant un cycle complet par besoin. Ces méthodes ont l'avantage de fournir un cadre strict permettant la réalisation d'un

projet pour lequel les spécifications sont justes, complètes et immuables. Cependant, l'utilisateur final peut ne jamais être présent dans les étapes de spécifications. De plus, il est possible que l'équipe de développement soit démantelée une fois le produit livré : le support technique du produit fini est alors pris en charge par une nouvelle équipe voire par une entreprise tierce, réduisant ainsi fortement ses possibilités d'évolution.

Les méthodes agiles [BBVB+01] sont apparues afin de corriger ces problèmes, présents dans la majorité des projets informatiques. Elles partent du principe que si le génie civil est capable de calculer à l'avance précisément les besoins et le coût de chacun de ses projets (*e.g.* une route, un pont), ceci est fortement moins envisageable lorsqu'il s'agit d'un projet informatique [1, 2]. Ces méthodes cherchent à remplacer le suivi d'un plan prédéfini par la recherche d'un retour d'information le plus tôt possible, afin de pouvoir corriger les spécifications. De cette manière, le produit évolue suivant les besoins réels des parties prenantes et non selon l'expression initiale de besoins estimés. Nous pouvons classer ces méthodes en deux grandes familles : celles définissant une gestion de projet, au sens large, telles que Kanban [Ōno88, And10], Scrum [SB08] ou Lean [Rie11] ; et celles indiquant également des techniques de développement, telles que l'eXtreme Programming (XP) [Bec00]. Les premières proviennent généralement du monde industriel, cherchant à la fois à optimiser l'utilisation d'une chaîne de production (Kanban), mais également à accélérer la résolution de problèmes sur cette même chaîne (alertes Andon de Kanban) ou sur le produit fini (*Feedback Lean*, *Scrum meetings*...). Les secondes sont plus orientées vers le développement, en imposant certaines règles d'écriture (Pragmatic Programming [HT99]) voire de contexte (développement par binômes dans la méthodologie XP).

La mise en œuvre de ces approches orientées sur le retour d'informations, nécessite la mise en place d'infrastructures permettant à la fois le déploiement de versions parmi les plus récentes, voire de versions d'essais, de certaines fonctionnalités (*rollforward*), mais aussi le retour rapide à une situation stable en cas de problème (*rollback*). Ces modes de développement ont permis le rapprochement entre l'équipe de développement et le client – à travers l'équipe de livraison – avec la volonté d'assurer une meilleure compréhension des exigences fonctionnelles et non fonctionnelles de ce dernier, ainsi que d'augmenter les chances d'acceptation du résultat final. Cependant, le fossé entre les équipes de développement et les équipes opérationnelles déployant le livrable sur leurs infrastructures restait toujours présent. Bien que l'équipe d'Assurance Qualité (*Quality Assurance*, notée ci-après QA) puisse se charger de valider le bon fonctionnement de l'application sur un ensemble d'infrastructures cibles ; celui-ci est fini, généralement réservé à ces tests et ne peut pas garantir à 100% un fonctionnement équivalent sur l'infrastructure de production. De plus, les opérateurs doivent prendre en compte les évolutions de l'application lors de son déploiement, allant de la simple mise à jour de fichiers à la modification de schémas de base de données. De fait, la gestion de projet a évolué vers la *livraison continue*, *i.e.* la diffusion automatisée des livrables,

1. `https://glyph.twistedmatrix.com/2014/09/ungineering.html`
2. `http://www.cs.utexas.edu/~EWD/transcriptions/EWD10xx/EWD1036.html`

puis vers le *déploiement continu*, *i.e.* le déploiement automatisé desdits livrables en production : tout nouveau livrable validé par les tests doit pouvoir être déployé en production avec un minimum d'effort et de temps.

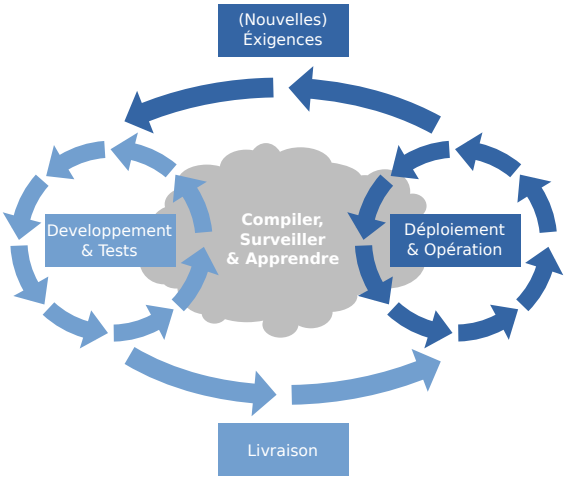

FIGURE 1.1 – Cycle de vie d'un projet selon la méthode DevOps
Schéma inspiré de [Dev15].

Cette méthode est la base du mouvement *DevOps*[Roc13], cherchant à faire travailler ensemble et efficacement les équipes de développement et les opérateurs de déploiement, comme le schématise la figure 1.1. Dans le cadre d'applications visant un grand nombre d'usagers, le déploiement automatique d'un livrable non final se déroule soit par l'inscription du client à un programme de bêta-test ou d'accès anticipé restreint (*Early Access*), soit par la sélection de la nouvelle version pour répondre aux requêtes d'un sous-ensemble aléatoire de clients. Dans le cadre d'un programme de bêta-tests, les utilisateurs qui souhaitent bénéficier de la dernière version disponible s'inscrivent au programme de déploiement. Ils sont alors notifiés du caractère potentiellement instable du livrable et sont invités à signaler le moindre problème. C'est par exemple le cas du programme *Aurora* de Firefox, livrant une version chaque nuit, ou des projets *Insiders* et *Preview for Developpers* de Microsoft. Le déploiement du nouveau livrable vers une sélection aléatoire de clients permet, par exemple, de remonter des informations utiles lors de la réalisation d'un test A/B (ou *split test*). Ce type de test consiste à fournir à chaque utilisateur, aléatoirement, l'une des versions d'un même produit ou d'une même fonctionnalité, qui diffèrent à plusieurs niveaux : esthétique, implémentation, etc.. La comparaison des retours d'information de la part des clients, par sondage ou par statistiques d'utilisation, permet de déterminer laquelle de ces versions est la plus efficace ou la plus rentable. Un exemple de test A/B est celui de Google [3] pour sélectionner la tonalité de bleu à utiliser dans son interface : 41 nuances de bleu ont ainsi été affichées aléatoirement aux visiteurs. L'une d'entre elle augmentait clairement le nombre de clics, et donc de revenus.

3. `http://www.nytimes.com/2009/03/01/business/01marissa.html`, pages 2–3

Cette méthode permet également de tester une fonctionnalité sans mettre en place l'intégralité des ressources nécessaires à son fonctionnement face au grand public. Par exemple, la messagerie instantanée WebRTC intégrée à Firefox, *Firefox Hello*, n'était disponible que pour 10% des utilisateurs de Firefox 35 [4], du 2 Septembre 2014 au 24 Février 2015, notamment pour confirmer que les serveurs de Mozilla pouvaient supporter la charge ajoutée par cette fonctionnalité.

L'approche à *micro-services* [5] est une mise en pratique emblématique du paradigme DevOps. Une application basée sur ce modèle est découpée en composants à gros grain, allant du regroupement de composants applicatifs traitant les mêmes éléments au gestionnaire de base de données. L'objectif est d'obtenir de bonnes performances en traitant un élément dans un minimum de composants différents, mais aussi de permettre la distribution et la duplication des composants de l'application, notamment pour faciliter un passage à l'échelle pour répondre à une montée en charge. Se basant sur la distribution de composants sur le réseau, cette approche met en avant la possibilité de défaillance des communications entre ceux-ci. Les équipes de développement doivent donc traiter ces cas d'erreurs. Un exemple d'une telle architecture est décrite dans le blog technique de Netflix [6], cette société ayant appliqué les méthodologies DevOps et Micro-Services à la lettre. L'approche *micro-services* peut être considérée comme un pont entre l'architecture multi-tiers et l'architecture à composants orientés service telle qu'employée dans cette thèse.

1.2 Problématique

Adopter la méthodologie *DevOps* nécessite la mise en place d'une chaîne de déploiement complète, de la phase de développement à celles du déploiement et de l'agrégation de données, après validation du livrable par les tests. Étant donné la large mise en œuvre de l'intégration continue, de nombreux produits ont été développés pour permettre aux développeurs d'atteindre rapidement la génération automatique de livrables après validation par les tests. Nous citerons par exemple Jenkins, Travis-CI, etc.. De leur côté, les opérateurs du déploiement ont également à disposition un certain nombre d'outils permettant d'orchestrer le déploiement d'un livrable, tels que Puppet, Chef, Ansible, etc..

Dans les modèles actuels, le déploiement d'un nouveau livrable peut se dérouler à deux niveaux : soit l'application entière est mise à jour et redémarrée, soit seul un ensemble d'artefacts est remplacé. Le premier cas est le plus simple à mettre en place, mais il ne permet pas de cibler les utilisateurs nécessitant cette mise à jour. Par exemple, si une fonctionnalité est présente dans le livrable mais n'est activée que pour certains utilisateurs, comme décrit précédemment pour Firefox Hello, alors même ceux ne pouvant utiliser cette fonctionnalité sont impactés. Dans le second cas, ce problème disparait, mais la technologie utilisée pour le remplacement de l'ensemble d'ar-

4. https://bugzilla.mozilla.org/show_bug.cgi?id=1083525#c9
5. http://martinfowler.com/articles/microservices.html
6. http://techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html

tefacts concerné peut nécessiter le redémarrage de l'application hôte. Dans le cas d'applications basées sur les composants, il peut être nécessaire d'instancier à nouveau les composants mis à jour, ce qui requiert un travail supplémentaire de gestion de composition. De plus, il est difficile de retourner des informations sur l'état de santé de ces nouveaux composants sans développer des sondes spécifiques. En effet, peu d'outils permettent de surveiller la consommation de ressources des différents composants d'une application, la plupart se limitant à un niveau de précision plus faible, *i.e.* à des mesures concernant la globalité de l'application. Parmi eux, nous retiendrons JRes [CvE98], qui définit une API de surveillance de la consommation CPU, mémoire et réseau d'un seul ou d'un groupe de fils d'exécution. Cette API a ensuite évolué ([CHS+05]) jusqu'à être spécifiée par le JSR-284 [Jav09]. Cette dernière n'était cependant pas encore intégrée dans Java 8. D'autres approches ont été proposées, afin de maîtriser la consommation des différents composants ou agents de l'application à travers des quotas, comme le proposent J-SEAL2 [Bin01] ou encore JAMUS [LSG02]. Ces techniques sont généralement intrusives, nécessitant un environnement d'exécution Java particulier, enveloppant les classes standards afin de leur ajouter des sondes. Enfin, bien que ces techniques détectent et empêchent les dépassements de quota d'un composant, elles ne permettent pas de déterminer si un composant fonctionne correctement.

Il apparaît donc important d'automatiser la gestion de nouvelles versions de composants et l'annulation de leur déploiement en cas de problème.

1.3 Objectifs

Pour répondre à cette problématique, nous avons développé une plate-forme de modélisation, d'exécution et de surveillance d'applications à composants. Ce travail a été mis en œuvre pour le compte de la société isandlaTech [7], sous le nom de *Cohorte* [8]. Elle est décomposée en deux grandes parties : *Cohorte Runtime*, se chargeant de l'exécution des hôtes des différents composants de l'application, et *Cohorte Composer*, se chargeant de leur instanciation.

Cohorte a notamment pour rôle de gérer la distribution des composants entre différentes machines physiques ou virtuelles. De cette manière, elle simplifie deux opérations de l'équipe opérationnelle : *a)* la répartition de charge, par duplication des ensembles de composants sur différentes machines, et *b)* la diffusion partielle d'une nouvelle version d'un composant, notamment pour le test A/B. En effet, chaque machine peut avoir une version différente d'un artefact fournissant un composant. Un répartiteur de charge frontal a pour fonction de sélectionner, aléatoirement ou non, la machine qui devra traiter les requêtes d'un client. Cette distribution permet de définir explicitement le placement nodal de certains composants de l'application. Ainsi, l'équipe opérationnelle peut indiquer la ou les machines sur lesquelles un composant doit être instancié, *e.g.* pour lui

7. `http://www.isandlatech.com/`
8. `http://cohorte.github.io/`

permettre d'accéder à un périphérique spécifique, tel qu'un équipement physique (capteur, actionneur, disque de données, ...). Un tel accès est cependant un facteur de risque pour la machine hôte. En effet, [PTS+11] montre qu'il est probable, dans le cas de matériel peu répandu, que le pilote permettant de dialoguer avec le matériel ne soit pas exempt d'erreurs. Ceci peut se traduire par des ralentissements, voire le crash, de la machine hôte. Le fait d'isoler le composant utilisant ce pilote du reste de l'application peut éviter la propagation de l'erreur.

Cohorte permet également de regrouper les composants dans différents conteneurs sur un même nœud, appelés isolats par la suite. Chaque isolat défini par l'équipe opérationnelle peut avoir une configuration propre. Cette capacité permet notamment de simplifier la surveillance de la consommation des ressources de la machine hôte par groupes de composants. Par exemple, un ensemble de composants Java peut nécessiter l'assignation de valeurs spécifiques aux propriétés de la machine virtuelle Java les exécutant, que ce soit pour limiter ou pour étendre ses autorisations de consommation de ressources. Ces conteneurs ont également pour but de restreindre la propagation d'erreurs, en particulier celles induisant l'arrêt impromptu d'un processus système. Dans un tel cas, la plate-forme se charge de diviser automatiquement les conteneurs fautifs afin d'en isoler le ou les composants erratiques.

Les isolations au niveau nodal (machine) et local (processus) simplifient le mécanisme de réjuvénation, aussi appelé *micro-reboot*, d'une partie des composants d'une application. Des exemples tels que [BOB92] (défaillance logicielle d'un missile anti-missile Patriot causant la perte de 28 militaires Américains) et [Fed15] (défaillance logicielle des contrôleurs du système électrique du Boeing 787, pouvant causer une perte de contrôle de l'appareil) montrent l'importance d'un tel processus. Dans le premier exemple, le redémarrage d'une partie de l'application de contrôle toutes les 8 heures aurait permis le bon fonctionnement du matériel de défense. Dans le second exemple, c'est un redémarrage matériel qui est rendu obligatoire afin de garantir que les unités de contrôle des générateurs électriques ne se mettent pas en sécurité. Dans ces deux exemples, les problèmes ont pour origine un dépassement d'entier (*integer overflow*).

Le principe d'isolation de composants permet à la plate-forme d'exécution de conserver active une partie de l'application, malgré la présence de composants instables.

De plus, cette plate-forme doit gérer l'intégration de composants implémentés dans différents langages. Pour ce faire, elle définit un modèle de composant générique, voire *basique*, dont les informations devront être converties, et éventuellement complétées, pour être utilisables par la plate-forme d'exécution du composant en question.

1.4 Contributions

Afin d'atteindre les objectifs décrits précédemment, notre contribution dans ce travail de recherche est la plate-forme Cohorte. Elle permet de décrire une application sous la forme d'une composition, d'isoler des composants et de les placer sur un nœud spécifique de l'infrastructure

des machines. La plate-forme se charge ensuite de l'exécution de l'application et du maintien de son intégrité. Cohorte a été partitionnée en quatre sous-projets :

— *iPOPO* est un modèle à composants orientés service, développé afin de retrouver dans le langage Python les concepts de génie logiciel promus en Java par l'OSGi Alliance. Le langage Python simplifie l'accès aux bibliothèques natives, utilisées notamment par *Cohorte Runtime*. Il a été adopté par une large communauté de développeurs et de scientifiques, pour des tâches allant de l'administration système aux calculs astronomiques en passant par la simulation physique. iPOPO est associé à Pelix, une plate-forme permettant le développement d'applications Python basées sur une architecture à services.

— *Cohorte Herald* est une bibliothèque de transmission de messages, simplifiant la communication entre les différents éléments de la plate-forme Cohorte. Elle permet également la consommation de services entre composants placés dans des isolats distincts.

— *Cohorte Runtime* est la plate-forme permettant de démarrer, surveiller et arrêter les isolats hébergeant les différents composants d'une application. Elle permet d'obtenir des informations de bas niveau sur les isolats tout en abstrayant les notions inhérentes au système d'exploitation hôte.

— *Cohorte Composer* gère la composition d'une application. Ce canevas calcule la distribution des composants entre les différents nœuds et isolats d'une application Cohorte afin de les répartir, soit selon les directives des opérateurs, soit automatiquement selon des critères d'origine et de fiabilité associés à chaque instance de composant.

1.5 Organisation du manuscrit

Le manuscrit est organisé de la manière suivante. Dans le chapitre 2, nous définissons l'état de l'art, en présentant les modèles à composants, les travaux concernant leurs communications distantes, ainsi que les travaux traitant de leur isolation. Nous y présentons également certains travaux dont les concepts, voire l'architecture, sont semblables à ceux de Cohorte mais sont utilisés à des fins plus ou moins proches. Dans les chapitres 3 à 6 nous détaillons les contributions développées pour les différents niveaux de cette plate-forme : le modèle à composants orientés service *iPOPO*, la couche de découverte et de communication inter-isolats *Cohorte Herald*, la plate-forme de gestion d'isolats *Cohorte Runtime* et enfin le gestionnaire de composants *Cohorte Composer*. Les projets ayant permis de valider la plate-forme Cohorte et leurs résultats sont décrits dans le chapitre 7. Enfin, nous concluons cette thèse en présentant les perspectives pouvant faire suite à ce travail.

CHAPITRE 2

ÉTAT DE L'ART

« Toutes choses sont bonnes ou mauvaises par comparaison. »

Edgar ALLAN POE, *Histoires Extraordinaires*, 1856

Sommaire

2.1 Introduction 10
2.2 Approche Orientée Composant 10
2.2.1 Définition 10
2.2.2 Implémentations 11
2.3 Approche Orientée Service 13
2.3.1 Définition 13
2.3.2 Implémentations 15
2.4 Approche à Composant Orienté Service 18
2.4.1 Définition 18
2.4.2 Implémentations 18
2.5 Distribution de services 20
2.5.1 Eclipse Communications Framework 21
2.5.2 ETHZ R-OSGi 21
2.5.3 Apache CXF DOSGi 22
2.5.4 OW2 RoSe et Fuchsia 22
2.5.5 Résumé 23
2.6 Isolation de composants 23
2.6.1 Isolation par processus 24
2.6.2 Isolation par conteneur 27
2.6.3 Isolation par machine virtuelle système 28
2.7 Canevas de distribution de composants 33
2.7.1 SCA 33
2.7.2 HighFrame 36
2.7.3 AIOLOS 38
2.8 Conclusion 39

2.1 Introduction

Les contributions de cette thèse couvrent trois domaines : *a)* les modèles à composants orientés service, *b)* la découverte et l'utilisation à distance de ces services, et enfin *c)* l'isolation des composants d'une application.

Dans ce chapitre, nous commençons par définir ce qu'est l'approche orientée service et ce qu'est un modèle à composants orientés service. Nous citerons les plates-formes correspondantes utilisées dans cette thèse, ainsi que quelques travaux proches. Nous présenterons les différentes implémentations de services distants, permettant aux services d'une application distribuée de communiquer entre eux. Nous évoquerons ensuite les travaux concernant l'isolation des composants d'une même application, et leurs différents objectifs. Enfin, nous présenterons plus en détail des canevas ayant une architecture comparable à celle de Cohorte, mais qui ont des objectifs différents.

2.2 Approche Orientée Composant

2.2.1 Définition

L'approche orientée composant, ou *Component-Based Software Engineering* (CBSE), consiste à développer des applications modulaires, décrites par des compositions d'entités fonctionnelles. Plusieurs définitions ont été proposées pour le terme *composant*. La plus couramment citée d'entre elles est celle de SZYPERSKI [SGM02] :

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*
>
> Clemens SZYPERSKI [SGM02]

Elle associe à un composant les caractéristiques suivantes :

— c'est une unité de composition ;

— ses fonctionnalités sont accessibles à travers des interfaces bien spécifiées ;

— il exprime explicitement ses dépendances vers d'autres composants ;

— il peut être déployé indépendamment dans la composition d'une application tierce.

Dans une autre définition, HEINEMANN [HC01] apporte la notion de modèle à composants :

> *A software component is "a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard."*
>
> George HEINEMANN [HC01]

Ces modèles spécifient la description et le cycle de vie d'un composant ainsi que son intégration avec d'autres composants, par composition. Ainsi, en respectant son modèle, un composant peut être déployé dans une nouvelle composition sans aucune modification.

À partir de ces définitions, nous pouvons donc considérer qu'un composant, représenté dans la figure 2.1, est une unité logicielle autonome se conformant à un modèle de composant. Ce modèle spécifie comment un composant décrit ses fonctionnalités ainsi que ses dépendances, qui seront utilisées pour l'intégrer dans une composition. Une séparation est alors clairement définie entre l'interfaçage du composant avec les autres éléments de sa composition et le code implémentant ses fonctionnalités. Ce code fonctionnel est vu comme une "boite noire", gérée par un *conteneur* permettant de l'adapter au modèle auquel se conforme le composant. Le conteneur prend en charge le cycle de vie du composant et ses interactions avec le reste de la composition. La liaison d'un composant à ses dépendances peut être mise en place de différentes manières par le conteneur. Généralement, elle est effectuée par inversion de contrôle (IoC), c'est-à-dire que le conteneur notifie le composant de la présence de la dépendance, au lieu de laisser ce dernier effectuer cette liaison. Le conteneur peut également directement injecter la référence vers une dépendance dans le composant. Cette opération peut être rendue possible après modification du code du composant ou par des approches telles que la programmation par aspect [SPDC06] ou la manipulation du code source ou intermédiaire. Ces techniques permettent notamment à un conteneur d'encapsuler du code patrimonial (*legacy*), en lui ajoutant des capacités d'adaptation à son environnement. Enfin, le conteneur traite également les propriétés et services non fonctionnels [DEM02] tels que la sécurité, la persistance, la surveillance des performances, etc..

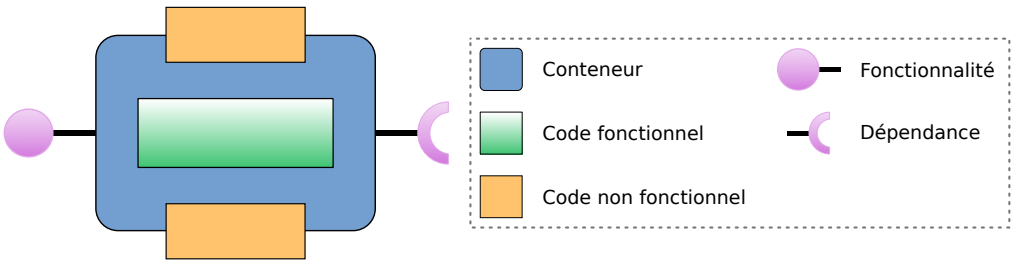

FIGURE 2.1 – Représentation d'un composant

2.2.2 Implémentations

Nous présentons ici quelques modèles à composants génériques, c'est-à-dire qui ne dépendent pas d'un domaine ou d'une application spécifique.

Le modèle *Enterprise JavaBeans* (EJB), a été publiquement spécifié par le JSR-19 [1] [Jav01]. Il permet de définir des composants qui sont ensuite exécutés dans des conteneurs d'EJB, eux-mêmes

1. JSR : Java Specification Request

constituant le serveur d'EJB avec des services non fonctionnels Java, *e.g.* traitant la persistance, les transactions XA, la sécurité, etc.. Un EJB est accessible à travers des interfaces pouvant être de type *remote*, permettant à un client d'utiliser ses fonctionnalités, ou *local*, afin de ne permettre la consommation de ses services qu'aux autres composants du même serveur. Il existe aussi une interface de fabrique *home*, grâce à laquelle un EJB est notifié par le conteneur des événements relatifs à son cycle de vie. Ces composants peuvent être avec ou sans état (respectivement, *stateful* ou *stateless*). Le conteneur d'un composant *stateful* doit garder en mémoire l'état de ce dernier et l'associer à une session client : cet état est alors restauré lorsque le client effectue une requête sur l'instance de l'EJB.

La spécification Corba 2.4 a défini le *Corba Component Model* [2] (CCM). Ce modèle est très proche des définitions précédentes : un composant CCM est géré par un conteneur, lui-même exécuté dans un serveur d'application. Un composant fournit une *interface*, ou référence, qui lui permet d'être utilisé par d'autres composants. Il est constitué de quatre types de ports : les facettes (*facets*), les réceptacles, et les sources et puits d'événements (*event sources* et *event sinks*). Les facettes correspondent à des spécifications de fonctionnalité, tandis que les réceptacles indiquent les dépendances à de telles spécifications. Le conteneur associe à un réceptacle la référence d'un composant fournissant la facette requise. Enfin, un composant possède des attributs configurables, correspondant aux propriétés le décrivant. Une version allégée du modèle, Light Weight CCM [3], a été spécifiée pour les dispositifs embarqués et à contraintes temps réel.

Fractal [BCL+06] provient quant à lui du monde académique. Ce modèle à composant est modulaire et extensible, et il ne cible pas un langage de programmation spécifique. L'implémentation de référence de Fractal est Julia [4], développée en Java. D'autres implémentations ont été proposées dans différents langages. Nous citerons pour exemples Cecilia [5], écrite en C, MIND [6], en C++, ou encore FractNet, ciblant la plate-forme d'exécution .NET. Les techniques utilisées pour construire l'instance d'un composant Fractal varient d'une implémentation à l'autre. Par exemple, Julia modifie le *bytecode* (code compilé) du composant pour y injecter directement les interceptions nécessaires au fonctionnement du conteneur. AOKell [SPDC06], une implémentation Java proche de Julia, utilise quant à elle la programmation par aspect pour associer différents éléments de contrôle au composant, au lieu de lui associer un contrôleur.

2. http://www.omg.org/spec/CCM/
3. https://staff.info.unamur.be/ven/CIS/OMG/lightweight%20component%20model.pdf
4. http://fractal.ow2.org/julia/
5. http://fractal.ow2.org/cecilia-site/current/index.html
6. http://mind.ow2.org/

2.3 Approche Orientée Service

2.3.1 Définition

L'architecture orientée service, *Service-Oriented Architecture* (SOA) est une approche selon laquelle une application est construite par l'assemblage d'entités découplées, fournissant leurs fonctionnalités à travers des services. Le terme *service* a été initialement défini comme suit par PAPAZOGLOU [PvdH03] :

> *Services are self-describing, platform-agnostic computational elements that support rapid, low-cost composition of distributed applications. Services perform functions, which can be anything from simple requests to complicated business processes.*
>
> Mike P. PAPAZOGLOU [PvdH03]

Dans cette définition, un service est une entité logicielle associée à la description de ses fonctionnalités. Cette entité est indépendante de la plate-forme d'exécution l'hébergeant. Un consommateur doit ainsi pouvoir l'utiliser en ayant uniquement connaissance de cette description, sans avoir à en connaître les détails d'implémentation. Les services sont ici présentés comme un outil permettant la liaison d'applications distribuées. Ils ne sont alors pas destinés à une utilisation intra-application.

Des précisions ont ensuite été apportées par ARSANJANI [Ars04] :

> *A service is a software resource (discoverable) with an externalized service description. This service description is available for searching, binding, and invocation by a service consumer. [...] Services should ideally be governed by declarative policies and thus support a dynamically re-configurable architectural style.*
>
> Ali ARSANJANI [Ars04]

Ainsi, la description du service doit être accessible depuis l'extérieur. Les consommateurs d'un service doivent alors être capables de le rechercher au moyen de sa description, de s'y lier et de l'utiliser. Cette définition précise également que les services d'une application devraient être régis par des règles déclaratives, et ainsi permettre l'usage d'un style architectural dynamique.

Tandis que [PvdH03] décrit le concept de service au sens large, [Ars04] définit le modèle d'interaction entre les fournisseurs et les consommateurs de ces entités. Un service doit ainsi :

— fournir une description de ses fonctionnalités et de ses propriétés à ses consommateurs potentiels ;

— masquer toute spécificité liée à sa plate-forme d'exécution, aussi bien dans sa description que dans son utilisation ;

— pouvoir être recherché selon sa description et être invocable par des consommateurs.

Nous pouvons en déduire qu'un service masque les détails de son implémentation en ne donnant accès à ses fonctionnalités qu'à travers un contrat. Nous retrouvons ici le principe d'encapsulation utilisé par les approches à composant. L'un des principaux intérêts de l'approche à service est le faible couplage entre les fournisseurs et leurs consommateurs : la liaison s'effectue selon un contrat, indépendamment du fonctionnement du fournisseur sélectionné. Un fournisseur de service peut ainsi en remplacer un autre, à condition qu'il respecte le même contrat.

L'approche orientée service définit également un modèle d'interaction à trois acteurs, décrit dans le schéma 2.2, afin de répondre aux besoins énoncés par [Ars04] :

— le *fournisseur* du service implémente l'ensemble des fonctionnalités indiquées dans sa description, appelée également *spécification*. Celle-ci peut être associée à des propriétés non-fonctionnelles permettant de décrire l'état du service.

— le *Registre* de services (aussi appelé *courtier* ou *annuaire* de services) est l'acteur auprès duquel le *fournisseur* enregistre son service. Son rôle est de renvoyer aux *consommateurs* la liste des services inscrits par les *fournisseurs*, filtrée selon les spécifications et les propriétés souhaitées.

— le *Consommateur* du service requiert les informations d'accès à un service auprès du *registre*, puis utilise les fonctionnalités de ce service conformément à sa spécification.

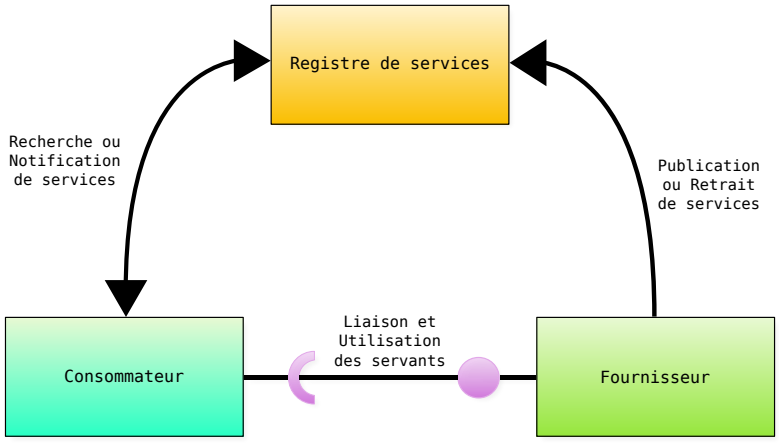

FIGURE 2.2 – Interactions dans le modèle SOA

Les interactions possibles entre acteurs sont les suivantes :

— *Publication* : le *fournisseur* inscrit un service auprès du *registre de services* ;

— *Retrait* : le *fournisseur* indique au *registre de services* qu'il ne fournit plus son service ;

— *Notification* : le *registre de services* notifie les *consommateurs* abonnés de l'inscription ou du retrait de chaque service ;

— *Recherche* : le *consommateur* demande au *registre de services* la liste des services répondant à une certaine spécification ;

— *Liaison* : le *consommateur* se connecte au service, selon les informations d'accès données par le *registre* ;

— *Utilisation* : le *consommateur* utilise les fonctionnalités du service, conformément à sa spécification.

La réalisation des étapes de publication, recherche et retrait d'un service peut être effectuée à n'importe quel moment durant l'exécution des fournisseurs et des consommateurs. Ainsi, l'approche orientée service favorise la liaison tardive (*late binding*) entre les fournisseurs et les consommateurs de services : leur liaison n'est pas déterminée à la conception ni au déploiement, mais en cours d'exécution. Cette propriété rend les applications plus dynamiques : leurs fonctionnalités ne sont mises en œuvre que lorsque leurs services requis sont présents et liés. De plus, l'utilisation d'une spécification, au lieu d'une implémentation, pour identifier un service permet de changer de fournisseur en temps voulu, ou d'utiliser un service de substitution lors du retrait de celui en cours d'utilisation.

2.3.2 Implémentations

L'adoption de l'approche orientée service a mené à la définition de spécifications et au développement de plates-formes, à différentes échelles. Nous présentons ci-après quelques exemples de ces productions.

Les *Web Services* [ACKM04], aussi notés *WS-**, en sont la première déclinaison la plus communément adoptée. Dans les faits, ce nom représente l'agrégation d'un ensemble de standards définis par les consortiums W3C [7] et OASIS [8] depuis 1998. Parmi les nombreuses spécifications de *WS-**, nous pouvons en retenir trois principales. La première est du format XML d'auto-description d'un service, le *Web Service Description Language* (WSDL). Ce langage permet en effet à un service de décrire ses fonctionnalités, ses propriétés, les types de données qu'il peut traiter, ainsi que ses modalités d'accès. *WS-** définit également un registre de services centralisé, *Universal Description Discovery and Integration* (UDDI), correspondant au registre de services présenté précédemment. Bien que cette spécification ait été finalisée, elle est généralement considérée obsolète, du fait de sa faible adoption par les éditeurs, fournisseurs et consommateurs de

7. World Wide Web Consortium : `http://www.w3.org`

8. Organization for the Advancement of Structured Information Standards : `https://www.oasis-open.org/`

services. Enfin, la communication entre services est normalisée par le *Simple Object Access Protocol* (SOAP). Elle est basée sur des messages XML, selon les indications de la description WSDL du service. La spécification n'indique pas de protocole de transport pour ces messages : bien que ceux-ci transitent généralement à l'aide du protocole HTTP, ils peuvent également être transmis via SMTP (protocole d'envoi de mails), TCP ou UDP (données brutes), etc..

Les *RESTful Web Services* [RR08], sont un style d'architecture visant à être plus léger que *WS-**. Ils s'appuient sur la thèse de Roy T. FIELDING [Fie00] sur le *REpresentiational State Transfer* (REST), une technique de gestion de ressources à travers des opérations dites CRUD (*Create*, *Read*, *Update*, *Delete*). Les services sont vus comme des ressources identifiées par un URI [9]. REST et RESTful n'étant pas des spécifications mais des styles d'architecture, ils n'imposent pas de protocole pour les mettre en œuvre. Cependant, la plupart des outils qui les utilisent s'appuient sur le protocole HTTP, en associant les opérations CRUD aux verbes HTTP (respectivement, *POST*, *GET*, *PUT*, *DELETE*). Les données sont alors transmises sous la forme d'objets JSON ou de documents XML.

UPnP [UPn08], *Universal Plug and Play*, est un ensemble de protocoles réseau permettant aux machines d'un même réseau de se découvrir et de fournir et consommer leurs services. Les services sont découverts au moyen du protocole *Simple Service Discovery Protocol* (SSDP). Celui-ci émet des paquets de découverte *multicast* suivant le protocole HTTPU, une version de HTTP fonctionnant en UDP. L'utilisation de paquets *multicast* évite d'avoir recours à un annuaire de services centralisé, tel qu'un serveur DNS ou DHCP, et permet ainsi de construire une architecture totalement décentralisée. De plus, les périphériques n'ont pas à être associés à une adresse IP sur leur réseau : UPnP utilise *AutoIP* pour leur en assigner une de type lien-local (`169.254.0.0/16`) automatiquement. La description des équipements et de leurs services ainsi que la notification des événements les concernant sont diffusées au format XML, suivant des schémas spécifiques à UPnP. Les services sont consommés à l'aide de messages *SOAP*, du standard *WS-**. Plusieurs extensions ont été proposées pour compléter UPnP. La plus répandue d'entre elles est l'extension multimédia UPnP AV, permettant à différents périphériques de partager de la musique, des vidéos et des photos. Ces périphériques peuvent avoir plusieurs rôles : *Digital Media Server* (DMS), pour fournir les fichiers, *Digital Media Renderer* (DMR), pour lire un fichier depuis un DMS, ou *Digital Media Controller* (DMC), pour contrôler un DMR. Cette extension est au cœur de la spécification DLNA [10], permettant les échanges de fichiers multimédias entre différents périphériques domestiques, *e.g.* entre un téléphone et une télévision.

DPWS [DM09], *Devices Profile for Web Services*, spécifie les fonctionnalités minimales des *Web Services* (*WS-**) à mettre en place pour permettre leur implémentation sur des périphériques embarqués. Ses concepts et ses objectifs sont similaires à ceux d'UPnP, c'est-à-dire découvrir les

9. Unique Resource Identifier
10. Digital Living Network Alliance – `http://www.dlna.org/guidelines`

périphériques d'un réseau et leurs services. Cependant, DPWS ne s'appuie que sur des protocoles *WS-** et ne spécifie pas de nouveau format de données.

Apache River [11] – initialement appelée *Jini* [Wal99] et développée par Sun Microsystems en 1999 – est une plate-forme Java permettant la description et l'exécution d'applications distribuées dynamiques. Elle est généralement définie comme étant une plate-forme à services orientés objet (*Service-Object-Oriented Architecture*, SOOA), du fait que chaque service est implémenté et fourni par un objet. La spécification du service est décrite par une ou plusieurs interfaces Java. Elle est associée à un dictionnaire de propriétés quand le service est enregistré dans un annuaire de services Jini, appelé *Lookup Service*. La plate-forme prend en charge la découverte des services et les appels distants à leurs méthodes : le consommateur d'un service n'a pas à être dépendant du protocole de communication sous-jacent.

OSGi [OSG12a] est une plate-forme Java SOOA modulaire. Nous l'avons choisie pour réaliser le développement de la plate-forme Cohorte. Sa modularité est assurée par les *bundles*, *i.e.* des fichiers archives Java (JAR) contenant du code, des ressources, et une description de leurs fonctionnalités et de leurs dépendances. Nous retrouvons ici la définition même du composant, telle qu'énoncée par SZYPERSKI. De plus, ces *bundles* ont un cycle de vie et peuvent être installés, démarrés, arrêtés, mis à jour et désinstallés à n'importe quel instant. La plate-forme offre ainsi les outils nécessaires à une exécution sans interruption de l'application. Comme *Apache River*, les services *OSGi* sont décrits par des interfaces Java et fournis par des objets. Contrairement à celle-ci, *OSGi* est une plate-forme centralisée, *i.e.* non-distribuée : les services ne sont enregistrés et accessibles qu'au sein de la même instance de la plate-forme, dans la même machine virtuelle Java. Cependant, la spécification *OSGi* définit les *Remote Services* afin de permettre la découverte et l'utilisation de services sur le réseau. Ces services distants sont inscrits dans le registre local et accessibles à travers un *proxy* (ou objet mandataire), afin que leurs consommateurs n'aient pas à traiter la problématique des appels à distance.

OSGi étant une spécification, il en existe plusieurs implémentations. Nous citerons notamment Apache Felix [12], Eclipse Equinox [13], Makewave Knopflerfish [14] ou encore Concierge [RA07], de l'École Polytechnique Fédérale (ETH) de Zurich. Bien que la spécification *OSGi* soit fortement liée aux concepts du langage Java, des implémentations ont été proposées dans d'autres langages. Par exemple, [EDH06] décrit les différentes techniques permettant de développer une plate-forme OSGi dans un langage de la plate-forme .NET (C#, VB.net, ...). Celix [15] et nOSGI [KDSH11] sont des plates-formes OSGi développées en C. L'éditeur Eclipse Orion repose sur Orion Core [16],

11. `http://river.apache.org/`
12. `https://felix.apache.org/`
13. `http://www.eclipse.org/equinox/`
14. `http://www.knopflerfish.org/`
15. `https://celix.apache.org/`
16. `https://wiki.eclipse.org/Orion`

qui implémente en JavaScript une petite partie de la spécification. Enfin, *Pelix*, l'une des contributions de cette thèse, en est une implémentation développée en Python. Elle sera décrite dans le chapitre 3.

2.4 Approche à Composant Orienté Service

2.4.1 Définition

Les exemples des plates-formes *Jini* et *OSGi* montrent qu'il est possible de développer une application non-distribuée en suivant l'approche à services. Le concept de composant orienté service ou *Service-Oriented Component Model*, noté SOCM, a été proposé par [Cer04], afin de lier approche à service et approche à composant. La définition du terme *composant* reste la même que précédemment : un composant peut fournir des services, donnant accès à ses fonctionnalités, et décrit ses dépendances à l'aide de spécifications de service. De cette manière, alors que les composants présentés précédemment sont généralement liés selon un câblage dépendant de leur modèle de composant ou de composition, voire simplement selon leur nom, les composants orientés service sont liés par leurs spécifications de service et par un filtre sur leurs propriétés.

Le développeur fournit le code fonctionnel du composant, tandis que le conteneur de ce dernier se charge de traiter des interactions avec le registre de services et donc de masquer la gestion du dynamisme de l'application. C'est également le conteneur qui doit prendre en charge l'enregistrement des services représentant les fonctionnalités du composant auprès du registre de services, selon son cycle de vie.

2.4.2 Implémentations

La spécification SCA [SCA11], *Service Component Architecture*, définit les représentations des composants et de la composition d'une application à base de composants orientés service. Chaque *composant* est une instance configurée d'une *implémentation*, pouvant être écrite dans différents langages, ciblant différentes plates-formes d'exécution. Cette spécification sera décrite plus en détails dans la section 2.7.1 et sera comparée à la plate-forme Cohorte dans ce même chapitre. Plusieurs implémentations de cette spécification existent, telles que OW2 FraSCAti [SMF+09] et Apache Tuscany [17], toutes deux écrites en Java. Cependant, SCA ne spécifie que la description et la mise en route d'une application, et n'indique pas clairement le comportement que doit avoir la plate-forme d'exécution face à ses contraintes de dynamisme. Des extensions telles que NaSCAr [Amé13] permettent cependant de traiter cette problématique.

L'écosystème OSGi a également proposé plusieurs modèles de composants orientés service.

17. `http://tuscany.apache.org/`

Le premier d'entre eux a été *ServiceBinder* [Cer04], qui a notamment été la preuve de concept de cette approche. La spécification OSGi Révision 4 a ensuite standardisé les *Declarative Services* [OSG05a] en s'inspirant de *ServiceBinder*. Dans ces deux modèles, les composants sont des objets Java instanciés par la plate-forme d'exécution. Ils sont décrits à l'aide de fichiers XML, présents dans le *bundle* OSGi fournissant leur implémentation, indiquant leur nom, leurs propriétés, les services qu'ils fournissent et ceux qu'ils requièrent. Chaque conteneur notifie le composant qu'il gère quand un service requis a été trouvé et lié à travers les méthodes indiquées dans sa description XML.

Blueprint [OSG13] a ensuite remplacé *Declarative Services* dans la Révision 5 de la spécification OSGi. Ce modèle décrit toujours les composants à l'aide d'un fichier XML, mais utilise la technique de l'inversion de contrôle (*Inversion of Control*, IoC) au lieu du mécanisme de notifications. Ainsi, le conteneur du composant prend en charge la gestion de ses dépendances et injecte ces dernières directement dans les champs indiqués dans la description. Cette technique évite au composant l'implémentation du traitement de ces tâches, potentiellement source d'erreurs. *Blueprint* est lui-même inspiré de *Spring Dynamic Modules* [18] (Spring DM). Ce dernier permettait de développer des applications Spring et de les déployer sur des plates-formes OSGi, tout en dispensant les développeurs d'avoir à interagir directement avec ces dernières. Le projet *Spring DM* a depuis été abandonné au profit de *Blueprint*.

iPOJO [Esc08] est un modèle à composant orienté service basé sur OSGi, mais ne faisant pas partie de sa spécification. Apparu avant *Blueprint*, il a été le premier modèle à imposer les approches IoC sur cette plate-forme. La description de ses composants peut être écrite dans un fichier XML ou bien à l'aide d'annotations Java, dans le code fonctionnel des composants. iPOJO supporte également les *composites*, c'est-à-dire des sous-ensembles de composants, utilisant un registre de services spécifique. Les composites sont hiérarchiques : un composite peut en contenir d'autres. À moins d'être explicitement exportés, les services des composants ne sont accessibles que dans leur composite. De même, pour accéder à un service d'un composite parent, celui-ci doit être explicitement importé par le composite fils. La particularité d'iPOJO est de manipuler le code intermédiaire (*bytecode*) de chaque classe implémentant un composant, afin d'y ajouter des méthodes de contrôle et d'injection de dépendances. Ces méthodes assurent notamment la protection contre les modifications concurrentes du composant et de son état, et permettent de masquer complètement la gestion des changements dynamiques des contrôles inversés du composant. De plus, iPOJO peut être étendu au moyen de son mécanisme de *handlers*. Chaque propriété ou aspect non fonctionnel d'un composant est géré par un *handler*, associé à son instance à travers son conteneur. Les *handlers* primordiaux prennent en charge l'injection des dépendances et l'enregistrement des services d'un composant. D'autres *handlers* d'extension peuvent être employés, par exemple, pour permettre à un opérateur d'accéder à certaines méthodes et propriétés du composant

18. `http://www.springsource.org/osgi`

via Java Management Extensions [Jav00] (JMX). Ces *handlers* d'extension sont configurés dans la description de l'instance de composant. Le modèle iPOPO, décrit dans le chapitre 3, s'inspire des spécifications d'iPOJO.

H-ubu [ELR13] est une version allégée du modèle iPOJO, développée en JavaScript. Son objectif est d'apporter l'approche à composant orienté service au code exécuté dans un navigateur web. De cette manière, les pages visitées contiennent un code adapté aux fonctionnalités du navigateur utilisé, telles que la gestion des images SVG, la présence d'un écran tactile, etc.. Les composants peuvent fournir des contrats, spécifiés à l'aide d'un objet JavaScript inerte, c'est-à-dire dont les méthodes n'ont pas de corps. Leurs dépendances peuvent être injectées directement dans un champ ou à l'aide de méthodes de rappels. Les services sont enregistrés dans un *hub*, servant à la fois de registre de services mais aussi d'intergiciel de communication inter-composants, basé sur des messages. Le *hub* est hiérarchique : il peut héberger des *sub-hubs*. À moins qu'il soit explicitement importé par le *sub-hub*, un service du *hub* parent n'est pas visible par les composants du *sub-hub*. De même, le *hub* parent n'a pas accès aux services du *sub-hub* qui ne sont pas explicitement exportés. Nous retrouvons ici l'approche adoptée par les *composites iPOJO*. Cette version du modèle ne supporte pas les propriétés non fonctionnelles, telles que les transactions, la persistance, etc..

Enfin, nous pouvons également citer la plate-forme SOSOC [DGR09]. Elle permet d'héberger, comme code fonctionnel d'un composant, des scripts écrits dans un des langages dynamiques supportés par la machine virtuelle Java, tels que JavaScript. Cette plate-forme se charge de l'exécution et des interactions de ce code avec le registre de services OSGi, afin d'y rendre accessibles ses fonctionnalités.

2.5 Distribution de services

Selon la définition de PAPAZOGLOU [PvdH03], l'approche orientée service a été initialement prévue pour lier entre elles des applications distribuées. Les *Web Services* suivent parfaitement cette définition.

Dans le cadre de cette thèse, nous nous concentrons sur le modèle SOOA (architecture à services orientés objet, *Service-Object-Oriented Architecture*), c'est-à-dire une approche à services liant entre eux les objets d'une même application, et plus précisément sur la spécification OSGi. Les plates-formes SOOA sont généralement centralisées et ne permettent pas de consommer les services fournis par une plate-forme sœur. Cette contrainte est levée à l'aide de fournisseurs de services distants : des entités (composants, plates-formes, ...) se chargent de découvrir et de se lier à ces services, puis d'enregistrer un mandataire pour chacun d'entre eux dans le registre de services local. La spécification OSGi définit les *Remote Services*, indiquant comment doivent être décrits et traités les services importés depuis une autre plate-forme, ainsi que les propriétés né-

cessaires à l'export d'un service. Plusieurs implémentations de cette spécification existent ; nous décrivons quelques unes d'entre elles dans les sections suivantes.

La consommation de services SOOA s'effectue à travers des appels de méthodes à distances, ou *Remote Procedure Call* (RPC), définis par [BN84]. Ce style architectural a été mis en œuvre par différentes plates-formes, telles que CORBA [19] à partir de 1991 ou encore DCOM [20] à partir de 1996.

2.5.1 Eclipse Communications Framework

Le projet *Eclipse Communications Framework* [21] (ECF) a été initié en 2004 par Scott LEWIS afin de permettre à des applications Eclipse de communiquer entre elles. Bien que le projet ait déjà été constitué d'un ensemble de *bundles* OSGi (Révision 3, à l'époque), il ne se basait pas sur les services mais sur le modèle de greffons (*plugins*) de la plate-forme Eclipse. Un greffon Eclipse est un *bundle* OSGi qui contient un manifeste descripteur `plugin.xml` définissant des composants à instancier. Ces composants sont associés à une spécification, et sont directement liés à leurs consommateurs par la plate-forme d'exécution Eclipse. Le support de la spécification des *OSGi Remote Services* a été implémenté dès 2008, soit juste après sa diffusion.

ECF est un cadriciel extensible. Un ensemble minimal de *bundles* permet de représenter un service distant, un service à exporter et leurs accès, ainsi que de traiter les notifications correspondant à leur cycle de vie. Les technologies de découverte, de consommation et de fourniture de ces services sont implémentées par les *ECF providers* (fournisseurs ECF), chacun fourni par un ou plusieurs *bundles*.

Une extension d'ECF a été proposée [22] afin de permettre aux applications et greffons Eclipse d'interagir avec des applications développées à l'aide de Cohorte, qu'elles soient développées en Java ou en Python.

2.5.2 ETHZ R-OSGi

R-OSGi [RAR07] est un cadriciel développé entre 2007 et 2011 par Jan S. RELLERMEYER, à l'ETH Zurich. À l'origine conçu pour distribuer des services OSGi entre différentes instances de la plate-forme Concierge, R-OSGi se base sur la Révision 3 de la spécification OSGi. De fait, ce projet n'implémente pas les *OSGi Remote Services*, mais a recours à des principes proches, tels que l'utilisation de propriétés de service pour décrire comment doit être exporté et comment a été importé un service distant. Par défaut, R-OSGi utilise un protocole de découverte spécifique, jSLP [23] (*Java Service Location Protocol*). Les appels distants sont quant à eux assurés par la

19. Common Object Request Broker Architecture
20. Distributed Component Object Model
21. `https://eclipse.org/ecf`
22. `https://wiki.eclipse.org/IPopo_Compatible_Providers`
23. `http://jslp.sourceforge.net/`

transmission de messages binaires, à travers une connexion TCP. Le contenu de ces messages est préparé à l'aide des mécanismes de *sérialisation* Java. Comme pour ECF, l'ensemble de protocoles de découverte et de transport supporté par R-OSGi peut être étendu à l'aide de *bundles* OSGi. Cependant, les protocoles supplémentaires officiellement proposés – HTTP, Mina et Bluetooth RFCOMM – utilisent tous le format de données spécifique à R-OSGi. Cette dépendance à un format spécifique à Java réduit pour ce projet les capacités d'interagir avec d'autres langages.

Une extension ECF est disponible afin d'utiliser R-OSGi en tant que fournisseur de découverte, d'export et d'import de services distants dans le cadre de la spécification *OSGi Remote Services*.

2.5.3 Apache CXF DOSGi

Apache CXF Distributed OSGi [24], généralement notée DOSGi, est une implémentation des *Remote Services* OSGi utilisant les *Web Services*. Elle est diffusée comme une extension du cadriciel CXF. Ce dernier permet à une application Java de fournir et consommer des *Web Services* SOAP ou *Restful*. Les services sont décrits par des contrats WSDL et sont inscrits soit dans un annuaire centralisé répliqué Apache ZooKeeper [25], soit dans un fichier local. Les appels de méthodes sont quant à eux assurés par des messages SOAP, sur le protocole de transport HTTP. De cette manière, les services exportés par cet outil peuvent être consommés par des applications fonctionnant à travers des *web services*, quel que soit leur langage d'implémentation. Bien que DOSGi offre la possibilité de modifier le contenu des messages à transmettre, elle ne permet pas d'utiliser d'autres protocoles de transport ni d'autres formats de message.

2.5.4 OW2 RoSe et Fuchsia

Le cadriciel RoSe [Bar12] a été développé par Jonathan BARDIN. Il est hébergé par le projet OW2 Chameleon [26]. Son objectif est de permettre aux développeurs et aux administrateurs de gérer dynamiquement les liaisons entre services d'une application distribuée selon une configuration. Cette configuration peut être définie dans un fichier, respectant un langage spécifique, ou bien programmatiquement. Comme ECF, RoSe est modulaire et peut être facilement étendu pour supporter de nouveaux protocoles de consommation de services. RoSe fournit nativement le support des protocoles SOAP avec Apache CXF, REST avec Oracle Jersey, et JSON-RPC avec Jabsorb. Contrairement aux cadriciels présentés précédemment, RoSe s'appuie sur une configuration, stockée dans un fichier ou dans un serveur annuaire, pour importer les services distants au lieu d'utiliser un protocole de découverte. De cette manière, l'administrateur de l'application a la maîtrise des liaisons entre les nœuds de son application. Cette approche est particulièrement

24. `http://cxf.apache.org/distributed-osgi.html`
25. `https://zookeeper.apache.org/`
26. `http://wiki.chameleon.ow2.org/xwiki/bin/view/Main/Rose`

intéressante dans le cadre d'applications distribuées sur différents réseaux, pour lesquelles les protocoles de découverte peuvent ne pas fonctionner.

Le projet RoSe a été abandonné au profit de Fuchsia [27] durant l'été 2013. Ce dernier a été développé comme un nouveau projet, reprenant les concepts de RoSe et les implémentations de certains protocoles de transport. Fuchsia apporte notamment le support de protocoles standards de découverte de services, tels que mDNS ou UPnP.

2.5.5 Résumé

Les plates-formes présentées sont globalement peu dynamiques. En effet, elles se basent généralement sur un annuaire statique, tel qu'un fichier de configuration, ou sur des protocoles n'autorisant que l'ajout et la suppression de services. Étant donné que les propriétés d'un service peuvent évoluer au cours du temps, qu'elles soient fonctionnelles ou non, il nous semble indispensable que l'application puisse être notifiée de ces modifications, et donc de disposer d'un événement de mise à jour de service. De plus, certaines de ces plates-formes sont fortement liées à un langage. Par exemple, R-OSGi et la plupart des transports fournis par ECF utilisent la sérialisation Java pour transmettre les arguments et le résultat des méthodes appelées. Elles ne sont donc compatibles qu'avec un langage exécuté dans une machine virtuelle Java.

2.6 Isolation de composants

L'extensibilité qu'apportent des plates-formes telles qu'OSGi est accompagnée d'une augmentation des risques de défaillances des applications en bénéficiant. En effet, les greffons d'extension peuvent être instables voire causer des fautes logicielles entraînant l'arrêt de l'application, réduisant ainsi sa qualité de service. Ce problème est principalement rencontré dans le cadre du déploiement continu, comme dans le paradigme *DevOps*. Dans le cas d'applications pouvant être étendues par des greffons provenant de tierces parties, il est également possible d'être confronté à l'exécution de code malveillant : intégré à l'application, un greffon a accès aux données de son hôte, voire à celles de la machine l'exécutant, selon les droits dont il dispose. Selon son objectif, il peut transmettre ou détruire ces données.

En réponse à ces risques, la technique la plus utilisée est l'isolation des greffons. Ainsi ils n'ont qu'un accès réduit au cœur de l'application hôte. Nous pouvons distinguer trois grandes catégories d'isolation :

— l'isolation intra-processus, où les greffons n'ont accès qu'à une partie de la zone mémoire de leur processus hôte. Cette méthode est généralement disponible dans les machines vir-

27. `http://ow2-chameleon.github.io/fuchsia/`

tuelles langages. Nous citerons par exemple les *Security Managers* [28, 29] de la machine virtuelle Java, ou encore les domaines d'application [30] (*AppDomains*) de la plate-forme .NET. Ce niveau d'isolation n'est pas traité dans cette thèse car il ne protège pas suffisamment l'application contre le crash du processus hôte de composants erratiques ;

— l'isolation locale, où les greffons et l'application hôte sont exécutés sur la même machine, voire dans le même processus ;

— l'isolation (ou distribution) nodale, où les greffons sont exécutés sur une ou plusieurs machines, différentes de celle exécutant le cœur de l'application hôte.

Dans tous les mécanismes d'isolation, il est nécessaire de répondre à deux problématiques : la communication inter-isolats des composants et l'impact de ces communications sur les performances de l'application. En effet, l'isolation de composants nécessite la mise en place de liaisons pour leur permettre d'interagir. Celles-ci peuvent être génériques ou optimisées pour le système sur lequel elles seront utilisées. Dans tous les cas, elles ajouteront un délai de transmission par rapport à un appel direct de méthode et réduiront plus ou moins la vitesse d'exécution de l'application dans sa globalité. Nous présentons ci-après quelques travaux concernant les isolations locale et nodale.

L'isolation locale consiste à utiliser les mécanismes de protection du système d'exploitation hôte pour éviter qu'une faute d'un greffon provoque le crash de l'application dans son intégralité, voire de la machine hôte. Nous admettons comme isolation locale tout mécanisme permettant d'exécuter un greffon dans le même système d'exploitation que l'application hôte. Les machines virtuelles, exécutant un autre noyau, sont considérées comme une technique d'isolation nodale, chacune d'entre elles étant un nœud.

Les deux premières approches d'isolation exposées ci-après permettent une isolation locale. Enfin, nous présentons quelques travaux concernant la distribution des composants sur différents nœuds à des fins d'isolation.

2.6.1 Isolation par processus

L'isolation par processus consiste à utiliser les mécanismes de protection mémoire des systèmes d'exploitation pour isoler les composants. Chaque composant est exécuté dans un processus différent de l'application hôte et n'a donc pas accès à (la totalité de) la mémoire du reste de l'application. En utilisant cette approche, il est cependant possible de partager un certain nombre de ressources (zones mémoires, fichiers, *sockets* réseaux, ...) entre l'application hôte et ses greffons.

Étant donné sa très large adoption dans le monde industriel, le langage Java a spécifié une méthodologie permettant à une application de se diviser en isolats. L'API *Application Isolation*

28. https://docs.oracle.com/javase/8/docs/api/java/lang/SecurityManager.html
29. http://net.uom.gr/Books/Manuals/whitepaper.pdf
30. https://msdn.microsoft.com/fr-fr/library/2bh4z9hs%28v=vs.110%29.aspx

est décrite dans la JSR-121 [Jav06]. Son objectif est de définir les modes de communication inter-isolats et les types de ressources qu'ils peuvent partager.

La *Cloneable JVM* [KOS+07] est une implémentation de la JSR-121. Il s'agit d'une version modifiée de la machine virtuelle Java J9 d'IBM, dans sa version pour Linux. Une machine virtuelle Java est complètement initialisée avant de pouvoir servir de modèle pour les isolats. Chaque isolat est un processus fils de ce modèle, créé à l'aide de l'appel système `fork()`. Cette fonction permet de créer une image du fil d'exécution (*thread*) de l'appelant, et utilise la méthode du *copy-on-write* pour dupliquer la zone mémoire du processus père. Cette copie *paresseuse* consiste à ne dupliquer une zone mémoire que si elle est modifiée par l'un des processus, père ou fils. Elle permet ainsi de créer rapidement un nouveau processus, pouvant accéder aux ressources de son parent mais ne pouvant pas modifier sa zone mémoire. La duplication des autres fils d'exécution est traitée par la *Cloneable JVM*, juste après l'appel à `fork()`, et juste avant de reprendre l'exécution du code Java dans la nouvelle JVM.

I-JVM [GTM+09] est une machine virtuelle Java spécialisée visant à intégrer un mécanisme d'isolation à la plate-forme OSGi. Contrairement à la *Cloneable JVM*, il ne s'agit pas d'une implémentation de JSR-121. Tous les isolats partagent la même zone mémoire, faisant partie du même processus. Dans la spécification OSGi, chaque *bundle* est associé à un *ClassLoader* Java spécialisé qui lui est propre. Le *ClassLoader* est utilisé par Java pour charger des classes ou des ressources, en les recherchant selon un mécanisme qui lui est propre (*e.g.* dans un fichier binaire ou JAR, selon une URL, ...) et, si besoin, en faisant appel à ses *ClassLoaders* parents. Cette approche permet à une plate-forme OSGi de contrôler les accès aux classes et ressources entre *bundles*. Pour cette raison, I-JVM isole chaque *ClassLoader*, donc chaque *bundle*. Son objectif est de protéger l'application contre des *bundles* malicieux ou instables [31]. L'isolation mémoire permet d'éviter aux premiers d'avoir accès aux autres *bundles*. Les seconds sont détectés selon leur consommation CPU ou mémoire.

Native Client (NaCl) [YSD+09] est un bac à sable, agissant à plus haut niveau que les conteneurs décrits dans la section précédente. Ciblant principalement les navigateurs web fonctionnant sur un processeur x86, son objectif est d'isoler leurs extensions ainsi que le code présent dans les pages visitées (JavaScript, ActiveX, ...). Contrairement aux travaux présentés ci-dessus, cet outil vise les processus natifs, sans interpréteur ni machine virtuelle (Java ou autre) faisant office d'intermédiaire entre le code à exécuter et le système hôte. Cette isolation se base sur deux niveaux de bac à sable. Le bac à sable *interne* (*inner sandbox*) est considéré comme un sous-domaine de sécurité, intégré à un processus. Il effectue une analyse statique du code à exécuter, détecte les comportements à risque et en interdit l'exécution. Il peut également interdire l'exécution d'un code cherchant à s'auto-transformer, car cette technique est généralement utilisée pour masquer

31. Comme le disait Pierre DESPROGES : *"Entre une mauvaise cuisinière et une empoisonneuse il n'y a qu'une différence d'intention."*

un comportement malicieux. Il s'appuie sur la segmentation mémoire x86 pour contraindre les zones mémoires à ne contenir qu'uniquement du code ou des données, afin d'éviter les failles de bas niveau telles que les dépassements de tampon (*buffer overflow*). Le bac à sable *externe* (*outer sandbox*) assure, quant à lui, la liaison entre le bac à sable et le système hôte. Il est capable de filtrer les appels application-système afin d'en interdire ou de renvoyer un résultat forgé. Il a donc pour objectif de protéger le système. De cette manière, NaCl peut protéger l'application hôte contre les fautes logicielles et peut également éviter l'accès complet à la machine hôte par du contenu provenant de sources distantes.

La plate-forme *Sandboxed OSGi*, proposée par GAMA [SdG11], permet de grouper différents *bundles* dans des isolats. L'objectif est d'isoler du reste de l'application les composants et *bundles* à la stabilité non mesurée, ou provenant de sources non fiables. Cette isolation est autonomique [KC03], supportant l'auto-réparation (*self-healing*). *Sandboxed OSGi* est en effet capable d'effectuer des redémarrages ciblés (*micro-reboots*) ou non (*reboot*) pour débloquer un composant ou tout simplement redémarrer un isolat s'étant arrêté inopinément. Le *micro-reboot* est en fait une simple action de mise à jour d'un *bundle*, impliquant son arrêt, son nettoyage puis son redémarrage. Cette action lui permet de réinitialiser son état et, potentiellement, de corriger simplement une défaillance due à une utilisation prolongée de ses composants (voir l'exemple de [BOB92]). Si cette opération n'est pas possible, *e.g.* si le *bundle* échoue à redémarrer ou reste bloqué, alors l'isolat l'hébergeant est complètement redémarré (*reboot*).

Deux implémentations alternatives de cette plate-forme ont été réalisées. La première version est mono-processus et s'appuie sur la *Multitasking Virtual Machine* [CD01] (MVM) pour isoler les différents *bundles*, utilisant une méthode proche de I-JVM. La communication entre les isolats est alors assurée soit par la *Link API*, définie par la JSR-121, soit par un protocole spécifique reposant sur des *sockets* TCP. Ce dernier est utilisé dans la seconde version de la plate-forme, qui s'appuie sur des machines virtuelles Java démarrées indépendamment. L'isolation est alors assurée par les mécanismes standards du système d'exploitation hôte, chaque processus ayant sa propre zone mémoire, non partagée.

Jasmin est une plate-forme développée dans le cadre la thèse de ATTOUCHI [Att14]. Elle étend la plate-forme MIND [32], une implémentation du modèle à composant Fractal [BCL+06], développée en C. Chaque application Jasmin est vue comme un composant MIND et gérée par un composant Jasmin racine (*Jasmin root*). Jasmin supporte deux niveaux d'isolation, par processus et par conteneur. Dans le premier cas, chaque application Jasmin est exécutée dans un processus qui lui est dédié. Dans le second cas, les applications sont séparées en groupes. Toutes les applications d'un même groupe sont exécutées dans le même conteneur (*cgroups*), et chaque application est exécutée dans un processus propre. Chaque conteneur héberge également un contrôleur Jasmin local. La plate-forme dans son ensemble est gérée par le *Jasmin root*, exécuté dans le système

32. `http://mind.ow2.org/`

hôte, en dehors de tout conteneur. Le partage de services entre les différents conteneurs est assuré par des mécanismes de communication inter-processus, *Inter-Process Communication* (IPC), tels que les *named pipes* ou les *sockets* UNIX. Ces mécanismes ne sont disponibles que pour les communications entre processus d'une même machine, et ne peuvent être utilisés sur le réseau.

2.6.2 Isolation par conteneur

Le plus bas niveau d'isolation au sein d'un même système d'exploitation est le conteneur, ou *bac à sable*, permettant de masquer à un processus tout ou partie des ressources du système hôte. L'isolat n'a alors accès qu'à certains, voire aucun, périphériques, et le système de fichiers qui lui est accessible est généralement virtuel. Étant donnée cette isolation forte vis-à-vis du reste du système hôte, ce processus peut être exécuté virtuellement avec des droits d'administration ne concernant que son environnement d'exécution.

Sous Windows, nous pouvons citer par exemple *Sandboxie* [33], un outil commercial existant depuis 2004. L'application est exécutée dans un bac à sable : tous ses appels système sont interceptés par l'outil afin qu'elle ne puisse modifier quoi que ce soit dans le système hôte. L'application a accès en lecture seule à tout le système, mais ce qu'elle écrit ne modifie que le bac à sable. De cette manière, des éléments sensibles du système tels que la base de registre sont facilement protégés. En revanche, cet outil ne permet pas de masquer le contenu du système de fichier hôte et repose sur l'isolation mémoire du système d'exploitation pour éviter qu'un processus dans le bac à sable ne lise ou modifie la zone mémoire d'un processus exécuté dans le système hôte.

Sous Linux, plusieurs générations d'outils sont apparues, chacune reposant sur les avancées de la précédente. Les *Linux Containers* [34] (LXC), disponibles depuis 2008, permettent de virtualiser l'environnement d'exécution d'un processus. Ce dernier n'a ainsi accès qu'à une partie des ressources de la machine hôte (processeur, mémoire vive, périphériques, système de fichiers), sans être capable de détecter cet état de virtualisation. Les LXC se basent sur les *control groups*, abrégés *cgroups*, du noyau Linux, permettant de cloisonner fortement les conteneurs, afin que ceux-ci ne puissent accéder directement à leurs voisins. Des mécanismes équivalents aux *cgroups* sont disponibles sur d'autres systèmes d'exploitation, *e.g.* les *FreeBSD Jails* [KW00] ou encore les *Solaris Zones*, faisant partie des *Solaris Containers* [Lag05].

Les *cgroups* ont notamment été mis en œuvre dans le cadre des travaux de LOUVEL [Lou11] portant sur la plate-forme ARMOR. Les conteneurs sont alors utilisés pour définir des quotas d'utilisation des ressources (CPU, mémoire, réseau, etc.) de la machine hôte. Ils sont également employés pour réserver ces ressources, afin de garantir leur disponibilité lors de l'utilisation d'un composant les consommant de manière importante, et donc de garantir la qualité de son service.

33. `http://www.sandboxie.com/`
34. `https://linuxcontainers.org/`

Docker [35] est un outil permettant de gérer et d'exécuter des applications dans des conteneurs. Démarré en 2013, ce projet se base sur LXC, sur les *cgroups* et sur des appels directs au noyau Linux. Docker simplifie notamment la définition des conteneurs, leur provisionnement, ainsi que les points de liaison entre conteneurs. Par exemple, il permet à plusieurs conteneurs de partager plusieurs dossiers du système de fichiers. Bien que largement adopté par les communautés Linux et *DevOps*, Docker a été *forké* en 2014 par CoreOS, pour créer le projet Rocket [36]. Ce projet vise principalement à sécuriser l'exécution des conteneurs, notamment en évitant d'utiliser un unique démon pour contrôler tous les conteneurs, comme le fait Docker.

2.6.3 Isolation par machine virtuelle système

L'isolation la plus forte pouvant être mise en place est la machine virtuelle système. En effet, les ressources réelles de la machine hôte sont partagées avec les machines virtuelles invitées. Ce partage est supervisé par le *Virtual Machine Monitor* (VMM), qui gère l'allocation et l'accès aux ressources de l'hôte. D'après [SN05], le VMM peut être d'un des trois types suivants :

— Natif : il est exécuté directement sur le matériel hôte. Les hyperviseurs KVM [KKL+07] (*Kernel Virtual Machine*) et Xen [BDF+03] font partie de cette catégorie.

— *User-Mode* : le VMM est exécuté, en mode privilégié, dans un système d'exploitation hôte. Le matériel est alors plus ou moins virtualisé, *e.g.* la machine invitée peut reconnaître le processeur utilisé, tandis que les autres ressources sont virtuelles. Oracle VirtualBox [37] et VMWare Player [38] sont des exemples de ce type de VMM. C'est également dans cette catégorie que l'on retrouve les émulateurs, tels que QEMU [Bel05].

— *Dual-Mode* : comme en *user-mode*, le VMM repose sur un système d'exploitation hôte, mais il dispose d'une partie exécutée plus profondément comme en mode natif. C'est notamment le cas d'Hyper-V [39], de Microsoft, dont l'hyperviseur est exécuté au plus bas niveau, mais dont le VMM est exécuté sur un système Windows.

[SN05] associe à sa définition des machines virtuelles systèmes un ensemble de cas d'usages, dont les suivants sont proches de notre problématique :

— *Isolation d'application* : cette utilisation a pour but d'augmenter la robustesse d'un système. Les applications ayant un comportement erratique ou malveillant sont cantonnées dans des machines virtuelles : ainsi, elles ne peuvent pas atteindre ni endommager la machine hôte ou les machines virtuelles voisines. Par exemple, si une application cause un *Kernel Panic*, cela n'impactera pas les applications dans les autres machines virtuelles et il sera possible

35. https://www.docker.com/
36. https://coreos.com/blog/rocket/
37. https://www.virtualbox.org/
38. http://www.vmware.com/fr/products/player
39. https://technet.microsoft.com/fr-fr/windowsserver/dd448604.aspx

de redémarrer rapidement la machine virtuelle impliquée voire de la replacer dans un état sauvegardé précédemment (*snapshot)*.

— *Renforcement de sécurité* : chaque machine virtuelle est vue comme un bac à sable, isolant chaque environnement système virtualisé de ses voisins. L'exemple donné dans [SN05] cible plutôt des machines virtuelles mono-utilisateur : les données de chaque utilisateur sont ainsi isolées à l'aide des mécanismes de virtualisation.

— *Environnement d'exécution et Applications patrimoniales* : il est probable qu'une application, voire un composant d'une application, dépende d'un environnement d'exécution particulier, *e.g.* d'une méthode désormais obsolète, présente dans une version précédente du noyau d'un système d'exploitation ou encore d'une bibliothèque incompatible avec celle nécessaire à l'exécution d'un autre composant. L'utilisation d'une machine virtuelle permet alors d'exécuter chaque pan de l'application dans l'environnement qui lui convient.

Les machines virtuelles peuvent également être transférées d'une machine hôte à une autre, en n'entraînant qu'une interruption de service minimale. Cette migration peut être effectuée *à froid*, *i.e.* en redémarrant intégralement la machine virtuelle sur un nouvel hôte, ou *à chaud*, *i.e.* en migrant une machine en cours d'exécution, tel que présenté par [CFH+05]. Cette capacité peut être utilisée si la machine hôte est instable ou en panne, *e.g.* si des erreurs de lecture de disque dur sont trop fréquentes. Un autre cas d'usage de cette capacité est la *consolidation* de machines virtuelles [ER15], *i.e.* l'adaptation de la répartition de machines virtuelles aux ressources disponibles sur des machines physiques. Cette opération peut s'effectuer dans les deux sens. Par exemple, plusieurs machines virtuelles peuvent être regroupées sur une même machine physique afin d'utiliser au mieux ses ressources ; à l'inverse, une machine virtuelle peut être transférée sur une autre machine physique afin de disposer de plus de ressources pour elle seule.

L'informatique en nuage (*cloud computing*) est apparue grâce à cette capacité de *consolidation*. Afin de rentabiliser leurs infrastructures, certaines entreprises telles que Amazon ou Google ont décidé de louer les machines peu voire non utilisées, afin d'héberger des applications dans des machines virtuelles. Ces infrastructures ne servent pleinement que pendant de relativement courtes périodes, telles que le *black friday* [40] pour Amazon. Le cloud s'est ensuite divisé en trois grandes catégories de services :

— SaaS (*Software as a Service*) : le fournisseur propose aux clients des instances d'une application et se charge intégralement de leur exécution. Salesforce [41] a été l'un des premiers acteurs dans ce domaine. L'exemple le plus courant est l'ensemble des suites bureautiques en ligne, *e.g.* Google Drive [42] et Microsoft OneDrive [43].

40. Journée de soldes de fin d'année très importante aux États-Unis et au Canada
41. `http://www.salesforce.com/`
42. `https://drive.google.com/drive/`
43. `https://onedrive.live.com/`

— PaaS (*Platform as a Service*) : le fournisseur propose une plate-forme d'exécution aux développeurs et aux opérateurs d'une application. Par exemple, le Google App Engine [44] permet de diffuser une application Web à travers l'infrastructure de Google, sur l'une des plates-formes supportées, *e.g.* Django en Python. Cette contrainte est courante mais n'est pas partagée par tous les fournisseurs PaaS : les services tels que Bluemix [45] d'IBM, Heroku [46], ... permettent de spécifier une plate-forme d'exécution personnalisée. Les problématiques de gestion de charge sont généralement traitées par le fournisseur, si les développeurs respectent certaines règles de bonne conduite (*guidelines*).

— IaaS (*Infrastructure as a Service*) : le fournisseur propose ici l'accès complet à des machines virtuelles. Les développeurs et les opérateurs ont ainsi un contrôle complet sur ces machines, pouvant aussi bien servir de machines de développement (compilation croisée, tests d'intégration, ...) que de plate-forme de production, permettant un contrôle beaucoup plus fin qu'un PaaS. Les services Amazon EC2 [47], Microsoft Azure [48] font partie de cette catégorie.

Les services de cloud cités précédemment sont *publics*. Il est toutefois possible de créer un cloud *privé* en mettant en place les ressources nécessaires dans sa propre infrastructure, ou un cloud *hybride* qui fusionne l'utilisation d'un cloud *privé* et des machines d'un ou plusieurs clouds *publics* afin de supporter une charge temporairement plus importante que la normale. La mise en place d'une telle infrastructure nécessite l'utilisation d'applications spécialisées, afin de gérer deux niveaux d'administration :

— les systèmes de gestion d'infrastructure à grande échelle, tels que OpenStack [49], Open Nebula [50], OpenShift [51], ... permettent de contrôler les machines virtuelles, de leur définition à leur exécution, en passant par leur déploiement.

— les orchestrateurs cloud, tels que Cloudify [52], Roboconf [PTD+15], ... permettent quant à eux de déployer une application à travers une infrastructure. Ils sont généralement capables de diriger des systèmes de gestion d'infrastructure à grande ou petite échelle (*e.g.* Vagrant [53]) afin de démarrer, configurer et arrêter des machines virtuelles selon leurs besoins. Enfin, ils peuvent s'appuyer sur des outils de gestion de configuration et d'administration, tels que

44. https://appengine.google.com/
45. https://console.ng.bluemix.net/
46. https://www.heroku.com/
47. https://aws.amazon.com/fr/ec2/
48. https://azure.microsoft.com/
49. https://www.openstack.org/
50. http://opennebula.org/
51. https://www.openshift.com/
52. http://getcloudify.org/
53. https://www.vagrantup.com/

Salt [54], Chef [55], Puppet [56], Ansible [57], ...

Le cloud a été présenté ici comme un outil visant à faciliter la gestion des montées et baisses de la charge subie par une application, tout en optimisant l'utilisation des ressources physiques disponibles. Il est également utilisé pour répliquer géographiquement les machines virtuelles et les données d'une même application entre différents centres de calcul (*datacenters*). Le premier objectif de cet usage est de limiter les périodes d'indisponibilités de service (*outage*) lors d'événements locaux, tels que les inondations et les coupures de courant dues à l'ouragan Sandy [58]. Son second objectif est de rapprocher les données et les services de leurs utilisateurs, notamment afin de réduire au maximum la latence perçue par ces derniers : cette approche est appelée *Egde Computing*. Les mécanismes de support de charge peuvent, dans ce cas, être traités par l'application elle-même [DELD06] en se basant sur des critères spécifiques (latence, ...). Enfin, le placement géographique permet également de contrôler les aspects juridiques associés aux données, *e.g.* les acteurs majeurs du cloud ont migré les données des clients ne résidant pas aux États-Unis hors de la juridiction de ce pays [59].

Les méthodes d'isolation présentées ici sont complémentaires, comme le montre la figure 2.3. Il est en effet possible d'isoler tout ou partie d'une application dans un conteneur, lui-même hébergé dans une machine virtuelle, déployée dans le cloud et répliquée géographiquement. Un composant d'une application peut ainsi être géré par une plate-forme d'exécution, telle que Java EE [60]. Cette plate-forme peut elle-même être exécutée dans un conteneur (Docker, ...) afin, par exemple, de simplifier la gestion de ses dépendances logicielles, de gérer des quotas d'utilisation de ressources ou d'isoler son espace mémoire et faire en sorte que ses défaillances n'influent pas directement sur le reste de l'application. Le conteneur peut être hébergé dans une machine virtuelle (VirtualBox, ...), afin d'augmenter l'isolation mémoire ou pour protéger le reste de l'application contre des défaillances systèmes graves (*Kernel Panic*, écran bleu, ...) dues à certains composants. Une machine virtuelle peut évidemment héberger plusieurs conteneurs. Elle peut être déployée dans plusieurs centres de données d'un ou plusieurs acteurs du cloud (Amazon EC2, Microsoft Azure, ...) afin de la protéger de risques environnementaux locaux (panne de courant, inondation, ...) ou encore pour réduire la latence du service perçue par le client final. L'exemple représenté par la figure 2.3 montre qu'il est possible que les différents composants soient isolés dans des conteneurs. Chaque conteneur peut héberger une ou plusieurs plates-formes d'exécution, selon

54. http://saltstack.com/
55. http://www.chef.io/
56. https://puppetlabs.com/
57. http://www.ansible.com/
58. http://arstechnica.com/information-technology/2012/10/hurricane-sandy-takes-data-centers-offline-with-flooding-power-outages/
59. https://support.twitter.com/articles/20172528
60. Java Enterprise Edition : plate-forme d'exécution des composant Enterprise Java Bean (EJB)

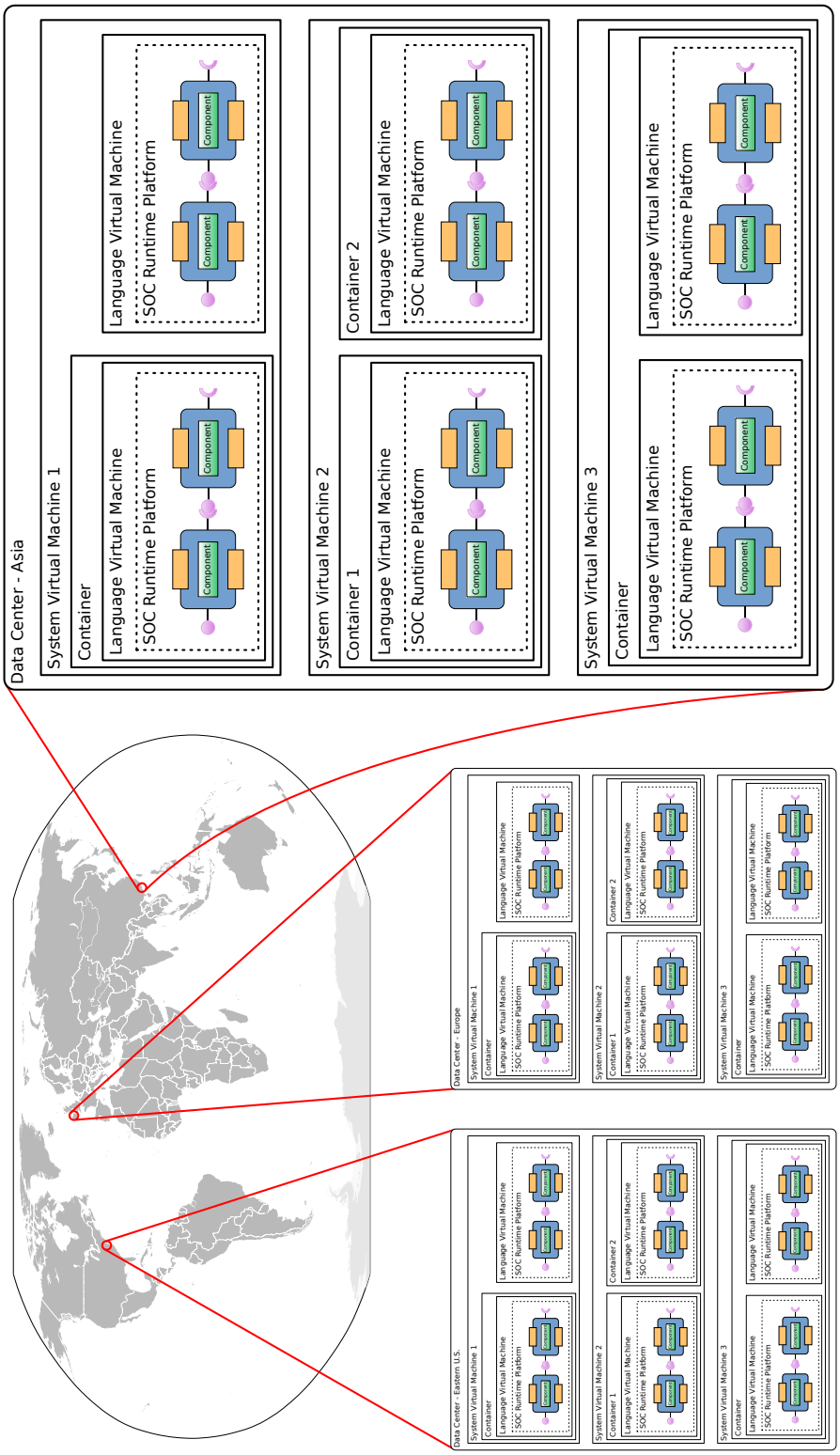

FIGURE 2.3 – Les différentes échelles d'isolation

le niveau de protection souhaité. Ces plates-formes ne nécessitent pas forcément de conteneur, comme le montre la machine virtuelle 1 de l'exemple. Cet exemple représente également la géo-réplication, ce groupe de machines virtuelles étant répliqué sur différents continents afin d'être au plus près de l'utilisateur final ou pour s'assurer une certaine résilience face aux pannes de centres de données.

2.7 Canevas de distribution de composants

Dans cette section, nous présentons plus en détail trois canevas de distribution de composants semblables à Cohorte Composer, présenté au chapitre 6. SCA [SCA11], cité précédemment, est un modèle de composition relativement proche du modèle employé par Cohorte. HighFrame [dSdRC14] est une plate-forme de développement d'applications réparties basées sur des composants. Enfin, AIOLOS [VSDTD12] est une plate-forme permettant la migration à chaud de composants d'une application mobile vers le cloud, dès que celui-ci est accessible. L'architecture de ces plates-formes est semblable à celle de Cohorte, cependant leurs objectifs sont différents.

2.7.1 SCA

SCA [SCA11], Service Component Architecture, est la spécification d'un modèle de programmation permettant de construire des applications basées sur une architecture orientée service (SOA). SCA fournit un modèle pour la composition de services et pour la création de composants à services, permettant également la réutilisation de fonctions d'applications existantes dans des composites SCA. SCA cible différents langages de programmation, différentes plates-formes de ces langages. Son extensibilité lui permet d'être utilisable avec un large panel de technologies.

Modèle

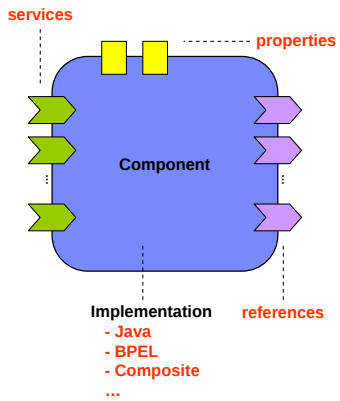

FIGURE 2.4 – Composant SCA

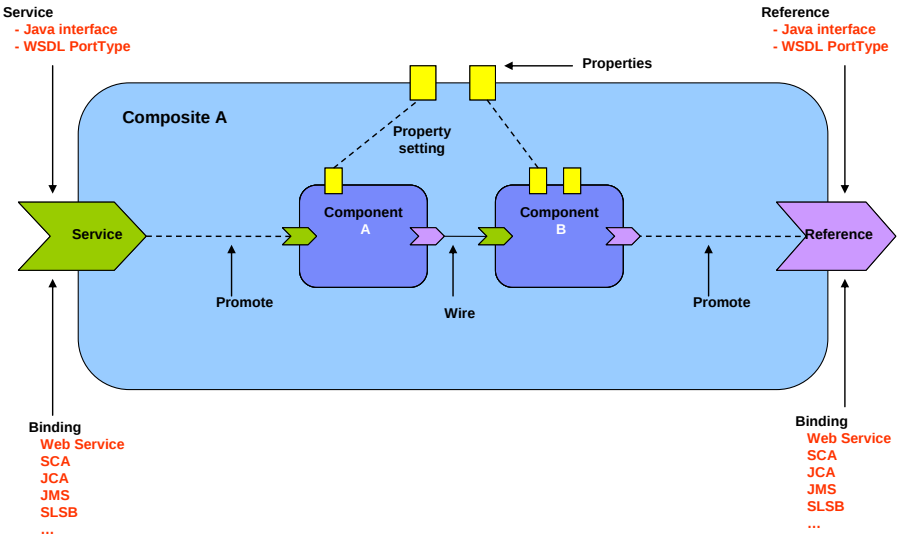

FIGURE 2.5 – Composite SCA

Le modèle d'assemblage de SCA, appelé *Assembly Model*, se décompose en quatre grands concepts : le *composant*, le *composite*, le *domaine* et les *wires* .

L'unité de construction d'une application SCA est le *composant* (Figure 2.4). Il s'agit d'une instance configurée d'une implémentation, cette dernière étant un code fournissant des fonctionnalités métier. Un composant peut fournir des services et peut avoir des *références*, c'est-à-dire des dépendances à des services d'autres composants. Il peut également porter des propriétés permettant de le configurer et lui permettant d'indiquer son état. L'ensemble des services, des références et des propriétés d'un composant sont les aspects configurables d'une implémentation qui représentent le type du composant.

Des composants peuvent être groupés dans un *composite* (Figure 2.5). Il s'agit d'un composant ayant pour implémentation un fichier de description contenant une liste de composants et leurs liaisons. Les liaisons entre composants sont décrites par des *wires* qui associent une référence à un service au sein d'un composite. Les services et les références de chaque composant ne sont visibles qu'à l'intérieur du composite s'ils ne sont pas promus, c'est-à-dire rendus explicitement visibles à l'extérieur. Étant donné qu'un composite est un composant avec une implémentation particulière, il est possible d'avoir des sous-composites, et donc de hiérarchiser la composition d'une application. Le principal intérêt est de pouvoir abstraire les détails de composition de certaines parties d'une application.

Le niveau supérieur d'une application SCA est le *domaine*, qui regroupe généralement l'ensemble des services fournissant une partie des fonctionnalités métier contrôlées par une seule organisation. Il est défini comme un composite particulier. Les liens entre les composants ne peuvent pas sortir du domaine. Pour communiquer avec l'extérieur, les composants doivent utiliser des

liaisons à adressage externe spécifique (URI d'un service SOAP, ...).

Implémentation

La spécification SCA définit la description d'implémentations de composants en langages Java, C++ et BPEL. Elle indique également la possibilité d'utiliser des langages de scripts (PHP, Javascript, ...) ou encore des langages déclaratifs (XQuery, SQL, ...). Il est également indiqué qu'un composite peut correspondre à une plate-forme d'exécution dans le langage d'implémentation de ses composants. Il est par exemple possible de définir des composants Entreprise Java Beans (EJB) et de les grouper dans un composite décrivant une plate-forme Java EE.

La liaison entre composants de langages différents est assurée par des Web Services : les interfaces des services concernés sont décrites au format WSDL 1.1, étendu de quelques informations spécifiques à SCA, telles que les SCA-Policies ou le support des callbacks. Les composants peuvent consommer d'autres types de services distants, mais aussi utiliser d'autres protocoles de transport. Il est ainsi possible d'associer une référence d'un composant à une URI décrivant un Web Service, de la lier à une queue JMS, ..., ou au nom (unique) d'un service d'un autre composant. Il est également possible de gérer des politiques d'accès au niveau de la définition des services ou des références.

Le support de ces différents langages et de leurs plates-formes dépend de l'implémentation de SCA par la plate-forme d'exécution. Les éléments d'une application SCA sont décrits dans des fichiers XML. Le modèle peut être étendu en définissant de nouveaux types de composant à l'aide de schémas XML.

Conditionnement et déploiement

Le domaine SCA représente la configuration complète de l'exécution d'une application, potentiellement distribuée sur plusieurs nœuds d'exécution. Tous ses artefacts sont décrits par des fichiers XML, et fournis par des *contributions*.

La spécification SCA définit quatre sources de contributions que doivent supporter les plates-formes d'exécution : elles peuvent être fournies dans un dossier (cas standard), sous la forme d'un bundle OSGi, d'un ZIP ou d'un JAR (et dérivés : WAR, EAR, ...). Ces contributions peuvent contenir des implémentations et des descriptions d'artefacts. La résolution des artefacts est réalisée selon l'espace de nommage de leur schéma XML ou de leur WSDL.

Les plates-formes d'exécution doivent supporter – selon la spécification – l'ajout, la mise à jour et la suppression de contributions, ainsi que l'ajout et la mise à jour de composites de déploiement. Dans le cas du composite représentant le domaine, il doit être possible d'y ajouter ou supprimer des éléments, aussi bien des composants que des liaisons (*wires*).

Un artefact peut être dans l'un de ces trois états : installé, déployé ou en cours d'exécution. La

plate-forme d'exécution doit refuser l'exécution de composants en erreur, et doit lever une exception, pendant le déploiement ou l'exécution, si la configuration du domaine ou de son contenu est invalide.

La spécification considère deux types d'erreurs. Celles détectées au déploiement, *e.g.* par analyse statique, pour lesquelles la spécification recommande l'annulation du déploiement, mais l'autorise à des fins de débogage. Les erreurs détectées à l'exécution, allant de celles non détectées par analyse statique aux Web Services requis mais indisponibles, doivent quant à elles être rapportées à l'opérateur du système. Les artefacts concernés par ces erreurs ne doivent pas être exécutés mais rester dans l'état *déployé*.

Plusieurs projets ont implémenté la spécification SCA, notamment FraSCAti [MRS11], NaSCAr [Amé13] ou encore Apache Tuscany [61].

2.7.2 HighFrame

HighFrame [dSdRC14] est une plate-forme de développement haut niveau pour des applications à composants. Elle est développée dans le cadre d'une thèse à l'Universidade Federal de Sergipe, Brésil, par Saulo Eduardo Galilleo SOUZA DOS SANTOS.

Cette plate-forme cherche à respecter le principe même de la programmation basée sur les composants : séparer au maximum le code fonctionnel du code technique. Elle prend en charge la génération des composants, leur déploiement, leur instanciation et leur liaison : le développeur n'a qu'à fournir le code fonctionnel des composants et l'architecture globale de l'application.

Modèle

HighFrame utilise le modèle défini par Fraclet [RM09], dont le métamodèle est décrit dans la figure 2.6.

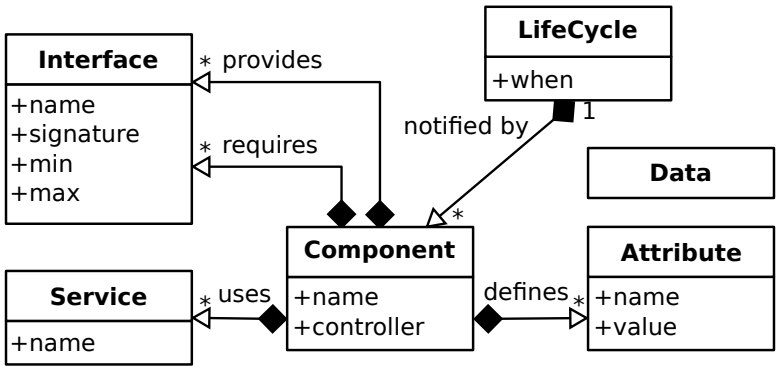

FIGURE 2.6 – Modèle de composant Fraclet

61. http://tuscany.apache.org/

Selon ce modèle, un *composant* est une entité identifiée par un nom, pouvant fournir et requérir des *interfaces*. Chaque interface est spécifiée par une signature, et identifiée par un nom. Un composant peut également utiliser les *services* fournis par la plate-forme d'exécution, différenciés des *interfaces* fournies par les composants. Un composant peut porter des *attributs*, dont la valeur est initialisée lors de son chargement. Enfin, le modèle indique qu'un composant peut être notifié des événements concernant son cycle de vie.

L'architecture d'une application HighFrame est décomposée en *sub-architectures*, contenant des définitions de composants et de leurs interfaces, fournies et requises. L'emplacement d'exécution des composants et leurs liaisons sont définis dans un fichier décrivant le plan de déploiement de l'application.

La plate-forme HighFrame est décomposée en cinq parties :

— Le *Manager* est le point d'entrée de la plate-forme. Il coordonne ses différents éléments et prend en charge la génération et le déploiement des composants selon les indications du développeur.

— Le *Component Generator* génère les composants à partir du code annoté fourni par le développeur au *manager*. Il peut exporter des artefacts pour différents modèles de composants.

— Le *Distributor* transmet les composants générés vers leur nœud hôte, selon le plan de déploiement indiqué par le développeur.

— Le *Deployment Agent* est responsable du déploiement, de l'instanciation et des liaisons des composants sur chaque nœud de l'application. Il prend également en charge la configuration de la couche de communication *InteropFrame*.

— Enfin, l'interopérabilité entre composants hétérogènes, *i.e.* entre composants de différents modèles, est assurée par *InteropFrame*. Elle permet aux composants d'utiliser, de manière transparente, les services de composants gérés par un modèle différent ou présents sur un autre nœud.

Implémentation

La description de l'architecture et le plan de déploiement de l'application sont décrits dans deux fichiers XML, au format *HighADL*. Ces fichiers sont générés par un outil graphique, simplifiant la définition d'une application HighFrame. L'architecture de l'application est validée lors de sa soumission au *Manager* avec le code fonctionnel des composants.

Le code fonctionnel fourni en entrée au *Manager* de HighFrame est écrit en Java et porte des annotations définies par Fraclet. Le *Code Generator* se charge de générer un composant pour le modèle cible (bundle OSGi, composant Fractal, composant OpenCOM, ...) indiqué dans le plan de déploiement. Cette étape se base sur des fichiers modèles, gérés avec l'outil Apache Velocity.

Ces composants sont ensuite fournis au *Distributor* qui se charge de les transmettre au nœud indiqué dans le plan de déploiement. Enfin, le *Deployment Agent* les instancie et les configure.

2.7.3 AIOLOS

AIOLOS [VSDTD12] est une plate-forme d'exécution développée par Tim VERBELEN, de l'Université de Gand, ayant pour objectif de simplifier le *Cyber Foraging* [BFS+02]. Le *Cyber Foraging* consiste à utiliser au mieux les ressources à disposition sur le réseau. Ainsi, un périphérique à ressources limitées, e.g. un téléphone, peut déporter des opérations consommatrices en ressources (temps de calcul, mémoire, ...) sur des serveurs spécialisés découverts sur le réseau local ou via une connexion Internet, tout en restant capable de les exécuter localement si aucun serveur n'est accessible. Cette technique permet de réduire la consommation électrique des périphériques embarqués, quand cela est possible, sans réduire leurs capacités opérationnelles hors connexion.

AIOLOS se focalise sur les méthodes de services. Celles déclarées comme déplaçables sont surveillées afin de calculer leur *profil*, c'est-à-dire l'historique des appels de méthodes contenant leur temps d'exécution, la taille de leurs arguments et la taille de leur valeur de retour. Le profil d'une méthode permet ainsi d'estimer s'il serait bénéfique d'opérer sa prochaine exécution localement ou sur un serveur distant. Cette estimation prend en compte la description des serveurs découverts. Celle-ci permet d'évaluer le gain potentiel sur le temps de calcul et le temps nécessaire aux transmissions de ses arguments et de son retour sur le réseau. Lorsqu'elle considère plus efficace d'exécuter une méthode sur un serveur, la plate-forme se charge de lui transmettre le composant fournissant le service associé ainsi que ses dépendances.

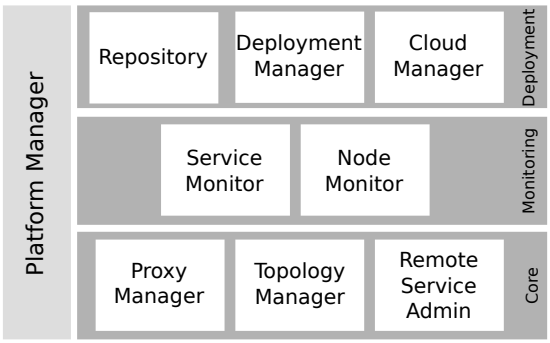

FIGURE 2.7 – Vue globale de la plate-forme AIOLOS

La plate-forme AIOLOS est découpée en trois grandes parties (voir figure 2.7), gérées par un composant *PlatformManager*. Ce composant permet de contrôler de manière unifiée les différentes couches de la plate-forme :

Le *Core* gère les appels distants en implémentant les *Remote Services* et la *Remote Service*

Administration spécifiés par OSGi. Le *Topology Manager*, défini dans la partie *Administration*, permet la liaison entre les différents nœuds de l'application, *i.e.* entre le téléphone et les serveurs détectés. La version implémentée par AIOLOS supporte également des politiques d'appels permettant, entre autres, de gérer la répartition de la charge sur les différents serveurs utilisables (*Round Robin*, ...). Les appels entre services sont quant à eux assurés par R-OSGi et Kryo.

La partie *Monitoring* collecte les informations qui concernent les services et les nœuds découverts. Un *ServiceMonitor* calcule le profil des méthodes de chaque service. Il est notifié de chaque appel distant par le *ProxyManager* du *Core*, indiquant le temps d'exécution ainsi que la taille des paramètres et de la valeur de retour associés. Le *NodeMonitor* collecte des données concernant le système de chaque serveur, telles que l'utilisation de CPU, la consommation mémoire, ... Ce composant n'est disponible que sur Linux car il s'appuie sur le dossier virtuel */proc*, spécifique à ce système.

Enfin, la partie *Deployment*, *via* le *DeploymentManager*, se charge de la recherche et du déploiement des composants, *i.e.* des bundles OSGi, et de leurs dépendances. Ces artefacts sont présents dans un dépôt géré par le composant *Repository*, basé sur le gestionnaire de dépôt spécifié par OSGi. De son côté, le *CloudManager* gère les machines virtuelles dans le cloud, si ce dernier est accessible. Si la charge des serveurs découverts est trop forte, ce gestionnaire peut démarrer de nouvelles machines virtuelles, pré-provisionnées d'une plate-forme OSGi et d'un agent de déploiement AIOLOS.

Implémentation

L'implémentation d'AIOLOS est basée sur OSGi et peut être exécutée sur les téléphones Android et sur les serveurs Linux. Pour cette plate-forme un composant est un bundle OSGi, pouvant consommer et fournir plusieurs services.

Afin de masquer l'utilisation d'un service distant, AIOLOS utilise les *Service Hooks* et les *Find Hooks* spécifiés par OSGi révision 4.2. Ceux-ci permettent d'intercepter les recherches de services et de renvoyer soit le service local, soit un proxy appelant le service distant, sans que le consommateur en soit notifié. Seuls les services dont l'implémentation a été déclarée avec l'annotation `@Offloadable` sont pris en compte. Ce proxy gère un délai d'attente : s'il est dépassé, l'appel distant est abandonné et le proxy utilise le service local. En cas d'erreur lors de l'appel distant, qu'elle soit au niveau réseau ou au niveau fonctionnel, le proxy utilise le service local.

2.8 Conclusion

Dans ce chapitre, nous avons présenté des travaux proches de nos contributions concernant les modèles à composants orientés service, la découverte et l'utilisation de services à distance et enfin l'isolation des composants d'une application. Nous avons ensuite décrit plus en détails des canevas

s'appuyant sur une architecture proche de celle de Cohorte, mais ayant des objectifs différents.

Le chapitre suivant présente la première contribution de cette thèse, iPOPO, un modèle à composants orientés service développé en Python.

CHAPITRE 3

CONTRIBUTION 1 : IPOPO

« Si l'implémentation s'explique difficilement, c'est une mauvaise idée.
Si l'implémentation s'explique facilement, c'est peut-être une bonne idée. »

Tim PETERS, *Zen Python (PEP-20)*, 2004

Sommaire

3.1 Introduction 42
3.2 Pelix : Architecture Orientée Service 43
3.2.1 Unité de déploiement 43
3.2.2 Services 45
3.3 iPOPO : Composants Orientés Service 46
3.3.1 Métamodèle 46
3.3.2 Fabriques de composants 47
3.3.3 Conteneur et handlers 49
3.3.4 Cycle de vie des instances 50
3.3.5 Liaisons inter-composants 52
3.4 Services distants 52
3.4.1 Architecture 53
3.4.2 Protocoles supportés 55
3.5 Services utilitaires 55
3.5.1 Pelix Shell 56
3.5.2 Service HTTP 57
3.5.3 Event Admin 57
3.5.4 Configuration Admin 58
3.6 Comparaison avec les modèles existants 59
3.6.1 Cycle de développement d'un composant 59
3.6.2 Liaisons entre composants 60
3.6.3 Propriétés extra-fonctionnelles 63
3.7 Conclusion 64

3.1 Introduction

OSGi et les modèles à composants orientés service comme iPOJO ont simplifié l'implémentation des applications dynamiques écrites en Java. OSGi a permis de normaliser la terminologie et la structure de plates-formes orientées service, en Java. iPOJO se base sur OSGi pour définir la notion de composant. Un type de composant iPOJO décrit les services qui seront requis et fournis par ses instances, ainsi que ses propriétés de configuration et sa fabrique, c'est-à-dire la classe Java implémentant le composant. Une instance de composant est une instance d'objet de cette classe fabrique, surveillée et dirigée par un gestionnaire d'instance iPOJO. Ce gestionnaire va notamment prendre en charge le cycle de vie de l'instance de composant qui lui est associée. La liaison entre les composants est décrite par des spécifications de services afin de réduire le couplage entre les composants fournisseurs et consommateurs.

Cette approche simplifie énormément le développement d'applications dynamiques : le développeur a la charge de décrire un type de composant et d'écrire les fonctionnalités métier de ce dernier, iPOJO se chargera de la mise en œuvre des composants de l'application. OSGi et iPOJO sont des technologies abouties, cependant ils ne sont disponibles que pour les applications Java. iPOPO a ainsi été développé dans le but d'offrir un sous-ensemble de leurs fonctionnalités dans le développement d'applications modulaires et dynamiques écrites dans le langage Python.

Le langage de programmation Python est largement adopté dans l'industrie logicielle. Il est présent dans des applications très populaires (Blender, Dropbox, ...). Il dispose d'un large écosystème de plates-formes Web (Flask, Django, ...) et de gestion (OpenERP, OpenStack, ...). Il est également utilisé par les acteurs majeurs de l'informatique en nuage, tels que Google (Youtube), Dropbox, etc.. Python est un langage à typage dynamique fort, de haut-niveau et orienté objet. Il a été initialement créé pour être l'intermédiaire entre le code natif C et les scripts *shell*, notamment en donnant accès aux méthodes des bibliothèques C avec une syntaxe très simple. Bien qu'il existe quelques modèles à composants pour Python, notamment la populaire *Zope Component Architecture* [1], aucun d'entre eux n'est orienté service et peu d'entre eux permettent de modifier dynamiquement les liaisons entre composants. Bien qu'iPOPO soit inspiré d'OSGi et d'iPOJO, les différences entre les langages Java et Python ont impliqué des approches différentes, ne serait-ce que pour respecter les pratiques communes de Python, définies dans la PEP-8 [2] et la PEP-20 [3] (appelée *Zen Python*).

Les objectifs initiaux de la définition du modèle étaient :

— les composants fournissant un même service doivent pouvoir être échangés à l'exécution et de manière transparente ;

1. `http://zope.org/`
2. `https://www.python.org/dev/peps/pep-0008/`
3. `https://www.python.org/dev/peps/pep-0020/`

— ils doivent pouvoir encapsuler des bibliothèques C de manière transparente, y compris si leur code source n'est pas ouvert ;

— les développeurs ne doivent avoir à écrire que du code fonctionnel dans un composant ;

— ils doivent avoir la possibilité de contrôler le comportement du composant,

— la gestion des composants doit être robuste pour que la plate-forme puisse être utilisée dans des applications à mission critique ou à longue durée d'exécution (*long-lived*).

La plate-forme iPOPO est divisée en deux grandes parties : iPOPO se charge des composants, tandis que Pelix se charge du registre de services, en essayant d'implémenter au mieux la spécification OSGi.

3.2 Pelix : Architecture Orientée Service

Pelix est la plate-forme à services constituant le socle d'iPOPO. Elle apporte la notion de *bundle*, unité de déploiement de la plate-forme, et implémente le registre de services spécifié par OSGi R4 [OSG05b]. Le nom Pelix est inspiré de celui de Felix, la plate-forme OSGi développée par la fondation Apache [4].

3.2.1 Unité de déploiement

L'unité de déploiement définie par OSGi est le *bundle* ; une archive Java (fichier JAR), contenant des classes Java, des ressources et un fichier de description, le *manifeste*. Dans la plate-forme Pelix, un bundle est un module Python, c'est-à-dire un unique fichier source (.py), un fichier *compilé* (.pyc, .pyo) ou un module natif (.pyd), chargé par l'interpréteur Python. À mi-chemin entre une classe et une archive Java, un module peut définir plusieurs constantes, classes et méthodes. La différence majeure entre un bundle Pelix et un module importé directement par Python est la gestion d'un cycle de vie : le bundle peut déclarer une classe d'activation pour être notifié de son passage d'un état à un autre. De cette manière, un bundle est capable d'allouer et libérer des ressources de manière autonome, ce qui n'est pas facilement réalisable avec un simple module. De plus, un bundle peut être installé, mis à jour et désinstallé en cours d'exécution, sans nécessiter le redémarrage de l'application dans son ensemble. Cette dernière conserve ainsi son état et limite son indisponibilité.

La figure 3.1 présente les différents états du cycle de vie d'un bundle, tel que définis par la spécification OSGi [OSG12a]. Quatre états sont possibles :

— `INSTALLED` : cet état est transitoire et indique que le bundle a été correctement analysé et chargé.

4. `http://felix.apache.org/`

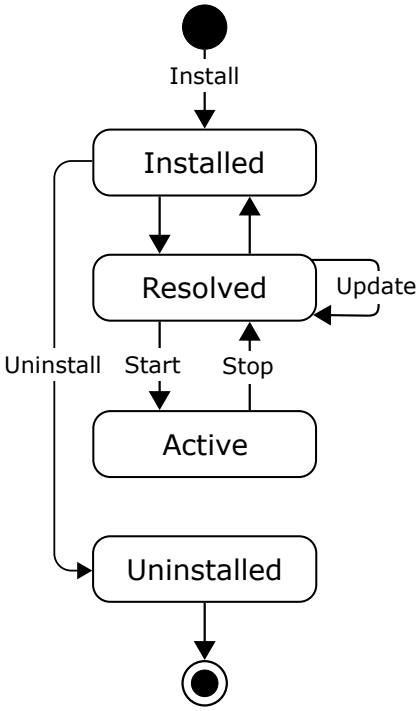

FIGURE 3.1 – Cycle de vie d'un bundle Pelix

— `RESOLVED` : le bundle est correctement installé dans la plate-forme, mais n'est pas actif. Son module est accessible, mais il ne peut fournir aucun service. Si le bundle était auparavant dans l'état `ACTIVE`, son activateur est notifié de l'arrêt du bundle et tous ses services sont automatiquement désinscrits.

— `ACTIVE` : l'activateur du bundle a été notifié du démarrage du bundle et a pu enregistrer les services du bundle. Dans cet état, le bundle peut enregistrer des services à tout moment.

— `UNINSTALLED` : le bundle est supprimé de la plate-forme et toutes ses références connues sont nettoyées, afin que l'interpréteur puisse le décharger complètement. Si le bundle était dans l'état `ACTIVE`, il doit être arrêté pour passer dans l'état `INSTALLED` avant d'être désinstallé. Une fois désinstallé, un bundle ne peut plus être réactivé tel quel : il doit être réinstallé au préalable.

Chaque bundle est associé à un contexte (*BundleContext*) tant qu'il est dans l'état `ACTIVE`. Le contexte est le lien entre un bundle et la plate-forme. Il sert notamment à enregistrer, à rechercher et à se lier à des services, à installer d'autres bundles, etc.. Il permet également à un bundle de s'abonner aux différents types d'événements de la plate-forme, concernant les bundles (changement d'état), les services (inscription, ...) ou l'arrêt de la plate-forme.

Selon la spécification OSGi, un bundle est décrit par son manifeste. Dans le cas de Java, il

s'agit du manifeste contenu dans son archive JAR. La plate-forme l'installant peut ainsi accéder à la liste de ses dépendances et déterminer s'il peut passer à l'état `RESOLVED` ou s'il doit rester dans l'état `INSTALLED`. En effet, lorsqu'un bundle Java est installé, ses classes ne sont pas chargées : il n'y a donc pas de tentative d'accès à des classes potentiellement absentes.

Dans le cas de Pelix, le bundle est un module Python, sans empaquetage. De ce fait, il n'est pas possible de lui associer simplement un manifeste et donc de décrire ses dépendances. L'installation d'un bundle Python correspond à son chargement dans l'interpréteur, c'est-à-dire à son analyse et à son exécution. Les dépendances du bundle sont alors chargées par les instructions *import* contenues dans son code : si l'une d'entre elles échoue, l'installation échoue. De même, si le contenu du module Python est incorrect (erreur de syntaxe, code opération inconnu, ...), l'installation échoue. Un bundle Python correctement chargé est donc, *a fortiori*, résolu et peut passer dans l'état `RESOLVED`.

3.2.2 Services

Un service Pelix suit la définition de la spécification OSGi. C'est une instance d'objet enregistrée dans un registre central, associée à la liste de spécifications de services implémentées par cet objet, ainsi qu'à un ensemble de propriétés. Chaque service est associé à un couple d'objets *ServiceRegistration* et *ServiceReference*. Le premier, *ServiceRegistration*, décrit l'enregistrement du service et il n'est accessible que par son fournisseur. En effet, cet objet est le seul permettant de modifier les propriétés du service et de le supprimer. La référence du service, *ServiceReference*, est quant à elle utilisée par ses consommateurs et par la plate-forme pour l'identifier. Elle donne accès, en consultation, aux propriétés du service et peut être utilisée afin de sélectionner un sous-ensemble de services parmi plusieurs avant de s'y lier et de les consommer.

La spécification d'un service est une chaîne de caractères. Dans le cas d'OSGi, il s'agit du nom complet d'une interface Java, qui n'a pas d'équivalent en Python ; Pelix autorise l'utilisation d'une chaîne de caractères quelconque, généralement enregistrée dans une constante. Les constantes indiquant les noms de spécifications ou de propriétés sont généralement définies dans des modules particuliers, qui n'ont pas à être chargés comme des *bundles*.

Les propriétés de service permettent de décrire à la fois les spécificités d'un service (la liste des fonctionnalités optionnelles implémentées, ...) ou son état actuel (état de connexion, ...). Dans le cas d'iPOPO, les propriétés d'un composant sont visibles dans les propriétés des services qu'il exporte.

Certaines propriétés, comme celles décrites dans la table 3.1, sont définies par la spécification OSGi pour décrire le service ou pour qu'il soit pris en compte par des services standards, tels que les *Remote Services* ou *EventAdmin*. La propriété standard `service.ranking` permet d'ajuster la priorité d'un service et donc sa position dans la liste des résultats lors d'une recherche de service. Cette propriété est notamment utile pour inscrire un service de *secours* : si les services spécialisés,

ayant une priorité importante, sont désinscrits ou tout simplement absents, un service de secours, ayant une priorité minimale, pourra être utilisé en dernier recours pour un fonctionnement en mode dégradé.

Propriété	Description
`service.id`	Identifiant unique du service dans la plate-forme
`objectClass`	Liste des spécifications implémentées par le service
`service.ranking`	Priorité du service : plus la valeur est grande, plus le service est prioritaire.
`service.exported.interfaces`	Liste des spécifications à exporter, si une implémentation d'*OSGi Remote Services* est présente

TABLE 3.1 – Extrait de propriétés spécifiées par OSGi

Lorsqu'un bundle souhaite consommer un service, il doit demander auprès du registre de la plate-forme la liste des services correspondant à ses critères. Ces critères peuvent être une spécification ou un filtre LDAP à appliquer sur les propriétés du service ou les deux. Un filtre LDAP est une chaîne de caractères contenant des critères liés par des opérations. Un critère LDAP est constitué d'un nom de propriété à tester, d'un opérateur de comparaison et de la valeur souhaitée. Les critères sont regroupés et liés par un opérateur booléen : ! (non), & (et), | (ou). Chaque critère ou ensemble de critères est entouré de parenthèses. Suivant la notation polonaise, l'opérateur booléen à appliquer aux critères est positionné au début du groupe. Par exemple, le filtre permettant de n'obtenir que les services qui ont une propriété `temperature` supérieure à 10° et dont l'unité est le *celsius* est le suivant : `(&(temperature>=10)(unit=celsius))`.

Enfin, le registre de services émet des événements lorsqu'il est modifié. De cette manière, il est possible pour un objet d'être notifié de l'inscription et la désinscription d'un service, ainsi que de la mise à jour des propriétés de ce service.

3.3 iPOPO : Composants Orientés Service

iPOPO est un modèle à composants orientés service. Chaque composant est une instance de classe Python, cette dernière étant appelée la *Component Factory* (fabrique de composant). Cette instance est gérée par un conteneur iPOPO qui l'associe à des *handlers* (gestionnaires) et gère son cycle de vie.

3.3.1 Métamodèle

La figure 3.2 représente le métamodèle d'iPOPO. Un composant est une instance d'une *classe fabrique*, décrite dans la section 3.3.2. Chaque *classe fabrique* est associée à un *contexte de fa-*

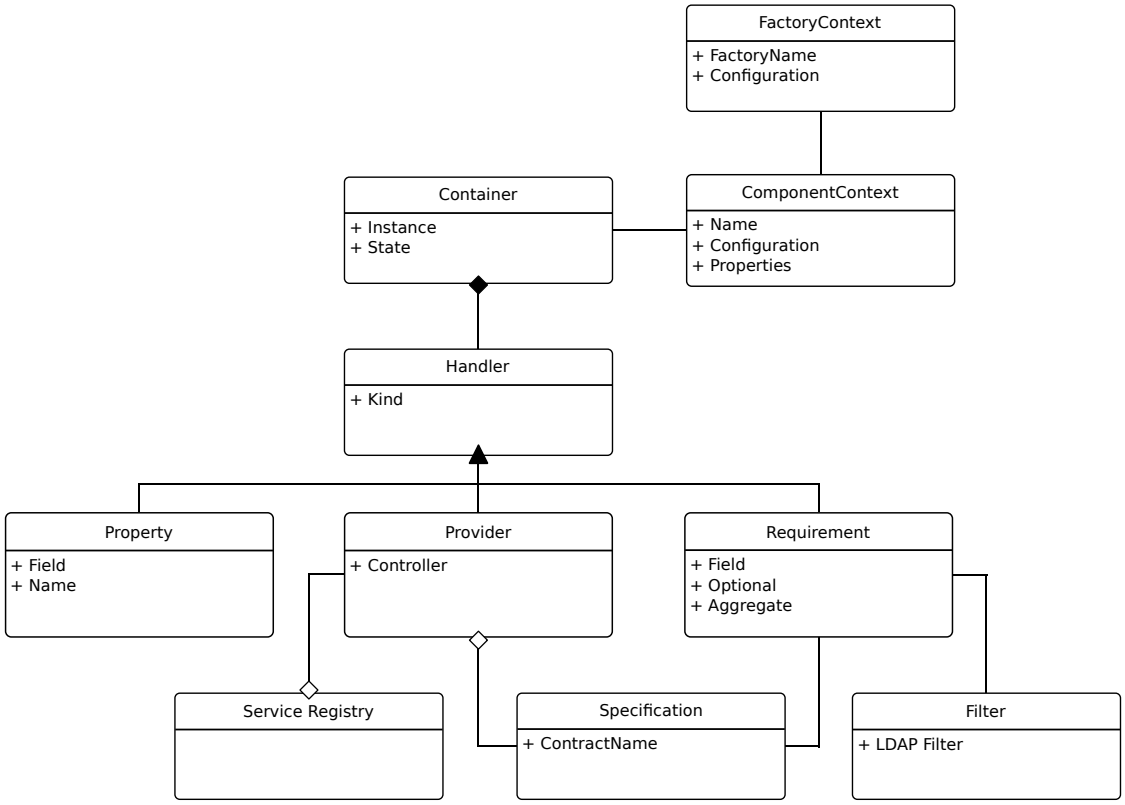

FIGURE 3.2 – Métamodèle iPOPO

brique décrivant la configuration par défaut de ses instances. À chaque composant est associé son *conteneur*, c'est-à-dire le gestionnaire qui s'occupe de son cycle de vie. Le *conteneur* lie l'instance à ses *handlers*, qui sont chargés des interactions entre le composant et la plate-forme. Ces éléments seront décrits plus en détail dans la section 3.3.3. Chaque conteneur est associé à un *contexte de composant*, lui-même associé au *contexte de fabrique* : il conserve la configuration spécifique à l'instance.

Les *handlers* présentés dans ce modèle sont ceux fournis par iPOPO pour qu'un composant puisse fonctionner : il est possible d'étendre le modèle. Le handler *requirement* permet de définir une dépendance par sa *spécification* et par un *filtre LDAP* sur les propriétés que le service doit déclarer. Le composant peut fournir des services et contrôler leur enregistrement dans le *registre de services* à travers des handlers de type *provider*. Enfin, le handler *property* permet de mettre à jour les propriétés du composant et des services qu'il fournit dès que le champ associé est modifié.

3.3.2 Fabriques de composants

Un composant iPOPO est défini par sa fabrique, c'est-à-dire par une classe Python modifiée par des décorateurs (*decorators*). En Python, un décorateur a pour rôle de modifier voire de rem-

placer le comportement de la classe ou la méthode décorée. Il s'agit d'une méthode ou d'un objet appelable qui prend la classe (ou la méthode) décorée en paramètre et en retourne soit une nouvelle version, soit une version modifiée. Dans le cas de la manipulation de classe par iPOPO, cette modification se limite à :

— l'ajout des champs où seront injectées les dépendances et les propriétés. De cette manière, iPOPO est assuré de pouvoir affecter des valeurs à ces champs, même si la classe ne l'autorise plus ensuite, *e.g.* dans le cas où elle définit le champ spécial `__slot__`[5], fixant ses champs.

— l'ajout des deux méthodes permettant d'intercepter la lecture et la modification de toutes les propriétés du composant.

— l'ajout du champ contenant le *contexte de fabrique*.

Chaque décorateur enregistre sa configuration dans le *contexte de fabrique* (*Factory Context)*, associée au type du *handler* qu'il décrit. Le décorateur spécial `@ComponentFactory` finalise la manipulation de la classe pour la rendre prête à être utilisée par iPOPO. Le *contexte de fabrique* ne peut alors plus être modifié.

Étant donné que les décorateurs Python sont exécutés lors du chargement de la classe, il est possible d'accéder aux informations de sa classe mère. Si elle hérite d'une fabrique de composant, alors elle hérite également d'une copie du *contexte de fabrique* de sa classe mère : seul le nom de fabrique doit être surchargé. Lors de l'héritage, il est possible pour la classe fille de refuser d'hériter de tout ou partie de cette description, afin de mieux contrôler son comportement.

```
1  @ComponentFactory(http.FACTORY_HTTP_BASIC)
2  @Provides(http.HTTP_SERVICE)
3  @Requires("_servlets_services", http.HTTP_SERVLET, True, True)
4  @Requires("_error_handler", http.HTTP_ERROR_PAGES, optional=True)
5  @Property("_address", http.HTTP_SERVICE_ADDRESS, DEFAULT_BIND_ADDRESS)
6  @Property("_port", http.HTTP_SERVICE_PORT, 8080)
7  # [...]
8  class HttpService(object):
9      # [...]
10     @Validate
11     def validate(self, bundle_context):
12         """
13         Component becomes valid
14         """
15         # ...
16     # [...]
```

Listing 1 – Exemple de fabrique de composant

5. https://docs.python.org/3/reference/datamodel.html#slots

Le listing 1 présente un extrait de la définition du composant serveur HTTP. L'exécution des décorateurs est effectuée dans l'ordre inverse de leur écriture. Une fois que l'interpréteur a analysé la déclaration de la classe (ligne 8), il appelle le décorateur écrit directement au-dessus de celle-ci (ici, `@Property` ligne 6), puis remonte jusqu'au décorateur `@ComponentFactory`. Les décorateurs `@Property` définissent les propriétés du composant et sont visibles dans le service qu'il fournit. Chacun d'entre eux indique le nom du champ où injecter la propriété, le nom de la propriété et sa valeur par défaut. Chaque décorateur `@Requires` décrit une dépendance du composant en précisant le champ où elle doit être injectée et la spécification de son service. Un filtre LDAP peut être défini pour sélectionner le ou les services à injecter selon leurs propriétés. Enfin, il est possible d'indiquer si cette dépendance est simple ou multiple, *i.e.* s'il faut injecter un ou plusieurs services, et si sa présence est optionnelle pour valider le composant. Le décorateur `@Provides` indique que les composants de cette fabrique fourniront chacun un service dès qu'ils seront validés. Il est possible d'ajouter un contrôleur pour que le composant puisse maîtriser plus précisément le moment où ce service devra être fourni. Enfin, `@ComponentFactory` inscrit le nom de la fabrique et finalise sa manipulation, tel que décrit précédemment. La fabrique peut maintenant être utilisée par iPOPO pour en instancier des composants.

3.3.3 Conteneur et handlers

Lors de l'instanciation d'un composant, iPOPO commence par retrouver la classe fabrique à utiliser, à partir du nom de type de composant demandé. Il en extrait le *contexte de fabrique*, décrivant les informations de base du composant, telles que la liste de ses *handlers* et leur configuration initiale. Ensuite, iPOPO crée l'instance du composant, *via* un appel sans argument au constructeur de la classe fabrique. Si cette étape réussit, il crée également le *contexte du composant* (*Component Context*), qui contient notamment la configuration active des *handlers*, celle-ci évoluant au cours de l'exécution du composant. Enfin, iPOPO crée le conteneur du composant, aussi appelé *Stored Instance*, et lui délègue la gestion intégrale du composant. Le service iPOPO ne conserve que l'association entre le nom d'instance du composant et son conteneur.

Le conteneur conserve la représentation de l'instance de composant qu'il doit gérer. Il garde ainsi l'accès au *contexte de composant*, au *contexte* du bundle qui l'a instancié et à l'instance du composant. La tache principale du conteneur est d'associer le composant à ses *handlers* et de gérer son cycle de vie.

Les *handlers* sont des objets créés par iPOPO auprès de *fabriques de handlers* (*Handler Factory*), et ce juste avant la création du conteneur. Il existe une fabrique par type de *handler* : selon la configuration associée à ce type dans le *contexte de fabrique*, cette fabrique crée autant de *handlers* que nécessaire. Si une fabrique est manquante, c'est-à-dire si un *handler* ne peut pas être créé, alors l'instanciation du composant est mise en attente : elle sera retentée dès que la fabrique manquante sera enregistrée. Une fois tous les *handlers* créés, ceux-ci sont associés au conteneur et

peuvent manipuler l'instance du composant. Durant cette étape, chacun d'entre eux peut injecter le code nécessaire à son fonctionnement. Le conteneur active ensuite ces *handlers* : durant cette étape, ceux qui le nécessitent s'abonnent aux événements de la plate-forme. Le conteneur va ensuite vérifier l'état des *handlers* à l'instanciation puis à chaque notification du conteneur par un *handler*, afin de déterminer l'état du composant et d'agir sur son cycle de vie si besoin.

Le fonctionnement d'iPOPO repose sur trois types de *handlers* :

— *properties* : ces *handlers* manipulent le composant pour que le conteneur soit notifié des changements de valeur des champs reflétant les propriétés du composant ;

— *service provider* : les *handlers* de ce type gèrent les services fournis par le composant ;

— *service dependency* : ceux-ci traquent et injectent les dépendances du composant dans les champs indiqués.

Les *handlers* de ces types régissent le cycle de vie d'un composant. Ils ne communiquent pas directement ensemble, mais par l'intermédiaire d'événements diffusés par le conteneur. Par exemple, si le composant modifie la valeur d'un champ de propriété, le *handler* de type *properties* notifie le conteneur. Ce dernier met ensuite à jour le *contexte de composant* puis notifie tous les *handlers* de ce changement de valeur. Ainsi, les *handlers* de type *service provider* peuvent alors mettre à jour les propriétés des services qu'ils ont enregistrés.

D'autres types de *handlers* peuvent être définis, *e.g.* pour traiter la sécurité, la persistance des données, les sessions, etc.. L'implémentation actuelle d'iPOPO ne fournit pas de *handlers* pour ces services non fonctionnels. Tous les *handlers* sont traités de la même manière et sont notifiés des mêmes événements. Les *handlers* de type *dependency* doivent cependant implémenter une méthode supplémentaire, utilisée afin de remplacer plus efficacement une dépendance disparue.

3.3.4 Cycle de vie des instances

Le cycle de vie d'un composant iPOPO est géré par son conteneur, lui-même s'appuyant sur les *handlers* pour déterminer si chacun d'entre eux est dans un état valide.

Comme le montre la figure 3.3, un composant peut se trouver dans l'un de ces quatre états :

— `INSTANTIATED` : le composant a été créé et a été couplé à son gestionnaire d'instance. Il n'est pas actif et ne peut donc fournir aucun service.

— `VALIDATED` : toutes les dépendances requises du composant ont été injectées, il est alors valide et peut donc inscrire des services.

— `KILLED` : l'instance du composant est en cours de suppression et ne pourra plus être valide. Toute référence au composant est supprimée et ses services sont désinscrits.

— `ERRONEOUS` : le composant a levé une exception durant sa validation. Il est alors invalidé, mais il n'est pas détruit. L'opérateur de l'application peut manuellement le supprimer ou retenter une validation.

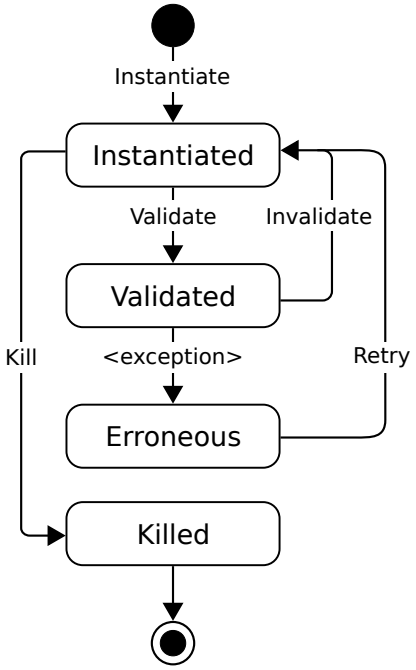

FIGURE 3.3 – Cycle de vie d'un composant iPOPO

Le composant peut être notifié des transitions de l'état `INSTANTIATED` à `VALIDATED` et vice-versa à l'aide des méthodes inscrites aux événements *validate* et *invalidate*. Il peut également être notifié de l'injection, de la mise à jour et de la suppression de chaque service injecté avec les événements *bind*, *update* et *unbind*.

L'état `ERRONEOUS` est un état intermédiaire, spécifique à iPOPO : le composant est instancié mais il ne peut pas être validé et n'est pas censé exécuter le moindre code fonctionnel. Le conteneur conserve la trace de l'exception ayant causé le passage à cet état, c'est-à-dire de l'exception qui a été levée pendant la validation du composant. Elle permet ainsi à l'utilisateur de déterminer si le composant échoue à cause d'une erreur de code, de contexte ou de configuration. Le composant n'étant pas dans l'état `KILLED`, son nom reste réservé, *i.e.* il n'est donc pas possible de recréer l'instance de composant par une nouvelle instanciation : l'utilisateur doit préalablement tuer explicitement l'instance actuelle. Cependant, iPOPO permet à l'utilisateur de tenter une nouvelle validation (*retry*) avec des propriétés mises à jour, c'est-à-dire avec une nouvelle configuration. Cette approche permet de corriger rapidement un composant mis en erreur par sa configuration ou son contexte. Le cas le plus courant ayant mené à la définition de cet état est l'instanciation de serveurs réseau : si le port d'écoute est déjà occupé, la configuration du serveur doit en définir un nouveau. Enfin, le fait de ne pas simplement laisser le composant dans l'état `INSTANTIATED` évite au conteneur de tenter une nouvelle validation à chaque événement traité par les *handlers* du composant : une succession de validations en échec peut notamment entraîner des fuites mémoire.

3.3.5 Liaisons inter-composants

Les liaisons entre composants sont gérées par des *handlers* de type dépendance (*dependency*). iPOPO fournit quatre *handlers* de ce type, cependant les développeurs peuvent en créer d'autres si nécessaire. Les *handlers* intégrés gèrent tous la notion de dépendance facultative : selon les paramètres du *handler*, le composant peut être actif même si aucun service ne correspond au besoin décrit. De plus, il est possible d'indiquer un filtre LDAP pour contrôler l'injection de services en fonction de leurs propriétés.

— `@Requires` gère les dépendances simples ou multiples (aussi appelées agrégées). C'est le *handler* le plus élémentaire : il indique que le composant dépend d'un ou plusieurs services fournissant une même spécification. Dans le cas d'une dépendance simple, le premier service répondant aux paramètres du *handler* est injecté ; dans le cas d'une agrégation, c'est une liste de services qui est injectée. Tant que le service injecté est présent, la dépendance est maintenue ; l'ordre des services dans la liste injectée est l'ordre de découverte et d'insertion : il ne respecte pas l'ordonnancement des services défini par la spécification OSGi.

— `@RequiresMap` est une évolution de `@Requires`, injectant un dictionnaire plutôt qu'un service ou une liste de services. La valeur de l'une des propriétés du service injecté sert de clé dans le dictionnaire. Dans le cas d'une dépendance simple, un seul service est associé pour une valeur de propriété ; dans le cas d'une agrégation, tous les services partageant la même valeur de propriété sont injectés dans une liste associée à cette valeur.

— `@RequiresBest` est également basé sur `@Requires`, mais il prend en compte l'ordre des services. Si un service avec un meilleur score (*ranking*) apparaît, il remplace celui qui était injecté. Ce *handler* ne gère pas les dépendances multiples.

— `@Temporal` autorise une période de temps durant laquelle la dépendance ne peut pas être satisfaite. Si le composant doit consommer cette dépendance, il est mis en attente jusqu'à ce qu'un service correspondant soit injecté ou que le délai d'attente soit dépassé, auquel cas une exception `TemporalException` est générée.

3.4 Services distants

iPOPO fournit une implémentation des services distants tels que les spécifie OSGi, appelée *Pelix Remote Services*. Sa version initiale s'inspirait des architectures de RoSe [Bar12] et de l'Eclipse Communication Framework [6] (ECF). La version actuelle a pour objectif principal d'être dynamique, c'est-à-dire de supporter de manière transparente l'ajout, la mise à jour et le retrait du support de protocoles de découverte et de transport à l'exécution.

6. `https://eclipse.org/ecf`

3.4.1 Architecture

Les *Pelix Remote Services* sont implémentés à l'aide de deux composants iPOPO formant son noyau, l'*export dispatcher* et l'*imports registry*, et s'appuient sur des fournisseurs de services de découverte et de transport. L'approche adoptée pour implémenter les *Pelix Remote Services* suit le patron de conception du tableau blanc (*whiteboard pattern*) : les composants du noyau consomment les services d'export, d'import et de découverte de services dès que ces derniers s'inscrivent. Ils sont aussi capables de gérer la disparition de ces services. La figure 3.4 représente les relations entre les différents services nécessaires à l'import et à l'export d'un service.

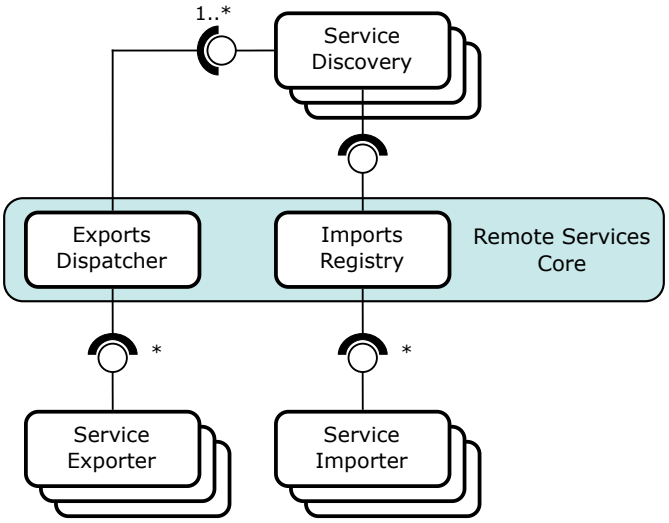

FIGURE 3.4 – Architecture des *Pelix Remote Services*

L'*export dispatcher* s'abonne auprès du registre de services Pelix aux événements concernant les services à exporter. Un service est à exporter s'il porte au moins l'une des propriétés suivantes :

— `service.exported.interfaces` : contient la liste des spécifications à exporter, qui doit être un sous-ensemble des spécifications du service. La valeur spéciale $\star$ indique que toutes les spécifications du service doivent être exportées.

— `service.exported.configs` : contient la liste des protocoles par lesquels doit être exporté ce service. Si cette propriété est vide ou absente, le service sera exporté par tous les protocoles disponibles.

Lors de l'inscription d'un service, l'*export dispatcher* est notifié par le registre de services et appelle tous les services exportateurs disponibles afin que chacun d'entre eux puisse créer un point d'accès (*endpoint*). Le point d'accès permet à l'exportateur de convertir une requête suivant le protocole de transport qu'il utilise en un appel de méthode du service souhaité. Il est associé à un identifiant, généré par l'exportateur, et à un nom, configuré dans les propriétés de service ou par l'exportateur, tous deux uniques pour une plate-forme. Une partie de la description du point

d'accès est spécifiée par OSGi, tandis que le reste dépend de l'implémentation de l'exportateur. Cette description doit pouvoir être transmise comme un dictionnaire de propriétés telles qu'elles seront vues dans les plates-formes qui importeront le service. Les propriétés d'export du service auront préalablement été transformées en propriétés décrivant un import. Un exportateur peut ne pas créer de *endpoint* dans le cas où le service à exporter ne répond pas à ses critères, *e.g.* s'il ne gère aucun des protocoles indiqués dans la propriété `service.exported.configs`. Dans le cas où un exportateur s'inscrit après l'apparition d'un service à exporter, il en sera tout de même notifié par l'*export dispatcher*. Enfin, dans le cas où des points d'accès ont été créés durant cette phase, l'*export dispatcher* notifie les services de découverte afin qu'ils émettent les événements indiquant la disponibilité du service selon leur protocole. Comme pour un exportateur, dans le cas où un service de découverte s'inscrit après la création des points d'accès, il sera tout de même notifié de leur existence afin qu'il puisse émettre les événements correspondants. Lorsqu'un service est mis à jour ou supprimé, l'*export dispatcher* notifie les exportateurs qui lui avaient créé un point d'accès puis les services de découverte. Dans le cas de la mise à jour des propriétés d'un service, il vérifie également si celui-ci est toujours exportable ou bien s'il l'est devenu.

L'*imports registry* est quant à lui notifié des événements que les services de découverte ont reçus selon leur protocole. Il se charge d'analyser et de valider la description du point d'accès puis transmet cette description aux services importateurs. Chaque importateur vérifie que les propriétés du service à importer lui correspondent et sont correctes ; il crée alors un mandataire (*proxy*) permettant d'accéder au service distant de manière transparente. Il inscrit ensuite ce mandataire dans le registre de services local, associé aux propriétés indiquées dans la description du point d'accès, c'est-à-dire les propriétés du service distant ainsi que des indications sur son import. Comme dans le cas de l'*export dispatcher*, quand un service importateur apparaît après la découverte d'un service, il sera tout de même utilisé afin de créer son mandataire. Le diagramme de séquence 3.5 présente quant à lui l'ordre des différents appels nécessaires à l'export et l'import d'un service.

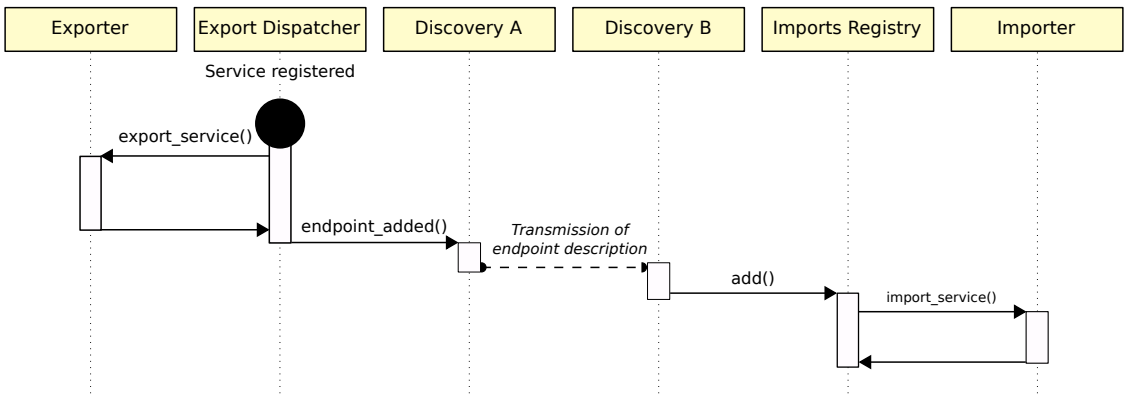

FIGURE 3.5 – Diagramme de séquence : export et import d'un nouveau service

3.4.2 Protocoles supportés

L'implémentation actuelle supporte nativement l'import et l'export de services accessibles *via* le protocole de transport XML-RPC, disponible dans la bibliothèque standard Python. À l'aide d'une bibliothèque externe, *jsonrpclib-pelix*, elle supporte également JSON-RPC. De plus, une variante de JSON-RPC, Jabsorb-RPC, est également supportée par les *Pelix Remote Services*. Ce protocole, qui est utilisé par RoSe, assure l'interopérabilité avec ce cadriciel. Il ajoute aux trames JSON-RPC des noms de classe Java, afin de transformer les objets JSON en objets Java, et plus précisément en *beans*.

En ce qui concerne les protocoles de découverte, les *Pelix Remote Services* supportent les fichiers de description EDEF (*Endpoint Description Extender Format*), définis dans la spécification OSGi Compendium [OSG05a], section 122.8. De cette façon, il est possible d'importer des services en utilisant des descriptions générées par ECF. Les *Pelix Remote Services* proposent également un protocole de découverte spécifique, basé sur des paquets UDP *multicast*. Il permet d'être notifié de l'ajout et du retrait de services, mais également de leurs mises à jour, ce qui n'est pas le cas de protocoles standards tels que (m)DNS-SD [CK13b, CK13a]. Des services de découverte s'appuyant sur d'autres protocoles tels que mDNS (*Zeroconf*), ou sur un annuaire tel qu'Apache ZooKeeper [7] ont été implémentés à titre d'expérimentation, notamment afin d'augmenter les capacités d'interopérabilité entre Herald et d'autres plates-formes. Ils n'ont cependant pas été finalisés, soit parce que l'interopérabilité est complexe à mettre en œuvre, *e.g.* les données enregistrées dans ZooKeeper par ECF sont au format binaire *serializable* spécifique à Java ; soit parce que le protocole ne permet pas la mise à jour de service.

L'implémentation des *Pelix Remote Services* est évidemment développée en Python, dans le cadre du projet Pelix/iPOPO. Cependant, une version de ce sous-projet a été développée en Java [8] afin de garantir l'interopérabilité entre ces deux langages, nécessaire au développement de la plate-forme Cohorte, décrite dans les chapitres suivants.

3.5 Services utilitaires

La plate-forme iPOPO fournit un ensemble de services utilitaires, dont certains sont issus de la spécification OSGi. Le respect de cette spécification diffère selon les services. Par exemple, *Configuration Admin* implémente l'API définie par la spécification OSGi, tandis que *Event Admin* en implémente les concepts, cependant il fournit une API différente.

7. `https://zookeeper.apache.org/`
8. `https://github.com/isandlaTech/cohorte-remote-services`

3.5.1 Pelix Shell

Le Pelix Shell permet à un utilisateur d'interagir avec la plate-forme. Bien que le projet Apache Felix ait proposé d'intégrer son sous-projet *Gogo Shell* [9] dans la spécification OSGi (*via* la RFC 147 [10]), cette dernière ne définit pas de shell. Le *Gogo Shell* s'appuie sur certaines capacités spécifiques à Java, telles que la récupération par réflexion des types des arguments dans la signature d'une méthode. De plus, le *Gogo shell* utilise une syntaxe avancée, proche de celle d'un shell Unix tel que `sh`. Par exemple, il supporte les *pipes*, permettant d'utiliser la sortie d'une première commande en entrée d'une seconde.

Nous avons préféré utiliser une syntaxe plus simple, n'autorisant l'exécution que d'une seule commande par ligne. La syntaxe utilisée par le Pelix Shell étant la même que celle d'une ligne de commande Unix simple, l'analyse de ces lignes est effectuée à l'aide du module standard Python `shlex` [11].

Le Pelix Shell s'appuie sur la notion de session afin de pouvoir répondre à plusieurs utilisateurs simultanés. Chaque session est associée à un *I/O Handler*, c'est-à-dire un objet permettant de dialoguer (écriture de lignes, demande d'entrée, ...) avec le client de manière transparente. Chaque implémentation d'un accès au shell doit fournir l'implémentation d'un *I/O Handler* correspondant à son protocole. Chaque session conserve également un ensemble de variables qui lui est propre, afin de sauvegarder et de réutiliser le résultat de commandes. Chaque session est créée par une interface du shell, dès qu'un nouveau client l'utilise. La plate-forme Pelix supporte, d'origine, trois interfaces pour le shell :

— Console : l'interface texte mono-client accessible depuis le terminal ayant lancé la plate-forme ;

— Remote : une interface accessible par le réseau, en connexion TCP brute. Elle supporte plusieurs clients en parallèle.

— XMPP : une interface accessible depuis un client XMPP. Le shell se connecte à un serveur XMPP (*e.g.* Google Talk) à l'aide d'un compte ou anonymement et peut traiter les commandes émises par plusieurs clients en parallèle.

Le Pelix Shell repose sur un composant noyau, le *Shell Core*, qui fournit un ensemble de commandes concernant la plate-forme (gestion des *bundles* et services, ...). Le *Shell Core* hérite lui-même d'une classe `ShellParser` se chargeant de l'analyse des commandes et de leur exécution, et qui fournit les commandes de base gérant les variables de la session.

Les commandes sont fournies par des services, détectés par le *Shell Core* dès leur enregistrement (*whiteboard pattern*). Chaque service implémente au moins deux méthodes : l'une indiquant

9. `http://felix.apache.org/documentation/subprojects/apache-felix-gogo.html`
10. `http://felix.apache.org/documentation/subprojects/apache-felix-gogo/rfc-147-overview.html`
11. `https://docs.python.org/3/library/shlex.html`

l'espace de nommage de ses commandes, afin d'éviter toute collision dans les noms de commande, et l'autre associant à chaque nom de commande la méthode à appeler pour la traiter. Les arguments donnés aux méthodes de traitement sont toujours la session du client suivie de la liste des paramètres détectés sur la ligne de commande.

3.5.2 Service HTTP

Le service HTTP fournit par Pelix est également inspiré de celui défini par la spécification OSGi [OSG13], cependant il ne suit pas exactement cette définition, pour la même raison que le shell de Pelix. La spécification dépend de Java et plus particulièrement de ses *servlets* [Jav11]. Le service HTTP de Pelix s'appuie sur le serveur HTTP présent dans la bibliothèque standard Python. Pelix fournit cependant une API proche de celle des *servlets* Java, notamment en donnant aux gestionnaires de requêtes un objet représentant la requête du client et un autre pour lui répondre. À titre de comparaison, un développeur utilisant l'API Python standard dispose uniquement de méthodes pour analyser l'ensemble des requêtes, *i.e.* il ne dispose pas d'outils permettant de router les requêtes vers la *servlet* associée à l'URL cible, et il n'a qu'un accès direct au flux pour y répondre, *i.e.* sans méthodes utilitaires.

À l'origine, OSGi ne spécifiait qu'une API d'enregistrement de *servlets* au service HTTP : ces dernières doivent explicitement s'inscrire auprès du service pour traiter des requêtes dont l'URL cible correspond à leurs filtres. La spécification OSGi R6 Enterprise [OSG15] apporte le support de l'enregistrement des *servlets via* le patron de conception *whiteboard pattern* : les *servlets* fournissent chacune un service associé à un ensemble de propriétés de configuration. Les services fournis par les *servlets* sont ensuite automatiquement détectés et utilisés par les services HTTP actifs.

Le service HTTP de Pelix implémentent ces deux patrons de conception (*whiteboard-pattern* et *listener*). Il permet de plus aux *servlets* d'être notifiées lors de leur association (et dissociation) avec un service HTTP. Durant cette notification, les *servlets* ont accès à certaines informations sur le serveur (port d'écoute, ...) et peuvent refuser de lui être associées. De cette manière, une *servlet* peut, par exemple, refuser d'être liée à un serveur accessible autrement que localement, ou encore connaître le point d'accès au serveur. Ces actions ne sont pas réalisables simplement avec l'API *servlet* Java, ni avec la spécification du service HTTP OSGi.

3.5.3 Event Admin

Le service *Event Admin*, qui est spécifié par OSGi [OSG13], définit un canevas asynchrone de publication-souscription (*publish-subscribe)* d'événements entre bundles ou composants.

Chaque événement est constitué d'un sujet (*topic*) et de propriétés. Le sujet est une chaîne de caractères définissant la nature de l'événement. Son format est généralement hiérarchique, chaque niveau étant séparé par une barre oblique (*/*, *slash*). La spécification OSGi définit un ensemble

de sujets afin qu'*Event Admin* propage les événements émis par la plate-forme. Par exemple, un événement ayant pour sujet `org/osgi/framework/BundleEvent/STARTED` est émis dès qu'un bundle a été démarré dans la plate-forme. Les propriétés associées à chaque événement sont un dictionnaire dont les clés et les valeurs sont spécifiées par documentation : il n'existe pas de moyen standard permettant de définir et de valider le format de ces données.

Un composant peut émettre un événement de manière synchrone ou asynchrone en consommant le service *Event Admin*. Celui-ci se charge de transmettre ledit événement à tous les services `EventListener` présents durant son émission et dont le filtre, indiqué en propriété de service, valide son sujet.

L'implémentation par Pelix de ce service suit la spécification OSGi à une nuance près : là où l'API Java utilise un objet représentant un événement, l'API Pelix le décompose en deux arguments, *i.e.* son sujet et ses propriétés. Cette approche permet notamment l'utilisation de gestionnaires d'événements distants : ceux-ci sont détectés à l'aide des *Pelix Remote Services* et sont notifiés si une propriété de l'événement indique que ce dernier peut être diffusé. De même, un pont, inspiré de [DT06], a été développé afin qu'un service Pelix puisse émettre et recevoir des messages suivant ce protocole, de manière transparente. Tandis que [DT06] utilise Java Message Service (JMS) comme protocole de transport, le pont implémenté pour Pelix s'appuie sur le protocole MQTT [12].

3.5.4 Configuration Admin

L'implémentation du service *Configuration Admin* par Pelix suit la spécification OSGi. Ce service permet de gérer les configurations persistantes des services de type `ManagedService`. Chacun de ces services est associé à un *identifiant persistant* (PID) unique. Comme son nom l'indique, la valeur de cet identifiant reste inchangée d'une exécution à l'autre de la plate-forme, contrairement à son identifiant dans le registre de services *service ID*, qui est variable.

Configuration Admin est capable d'enregistrer des fichiers de configuration dans un répertoire donné, en fonction des appels effectués par les différents services. Lors de son démarrage, *Configuration Admin* charge toutes les configurations présentes dans ce répertoire et notifie pour chacune d'entre elles le `ManagedService` dont le PID lui correspond. Il est possible que la configuration appliquée à un service ait été enregistrée par un service ou un composant différent de celui ciblé. Cette approche s'appuie sur le *whiteboard pattern* et permet ainsi aux services d'être configurés sans avoir à connaître ni l'emplacement de leur configuration ni la technologie de stockage utilisée (fichier de configuration, base de données, ...).

Dans son implémentation, Pelix enregistre les configurations des services dans des fichiers JSON, chacun ayant pour nom le PID du service à configurer. Pelix est capable de surveiller ces

12. Message Queuing Telemetry Transport
`http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html`

fichiers afin de prendre en compte leur mise à jour par modification extérieure (par l'opérateur, par un outil tiers, ...). Pour cela, nous avons développé un équivalent du service *FileInstall* [13], développé par Peter KRIENS puis maintenu par le projet Apache Felix. Le service *FileInstall* n'est pas défini par la spécification OSGi. Ce service scrute les répertoires indiqués dans les propriétés de la plate-forme : si un fichier est créé, modifié ou supprimé, il en notifie les services `ArtifactInstaller` associés à ce type de fichier. Dans son implémentation Java, *FileInstall* scrute les bundles OSGi (fichiers JAR), qu'il peut installer, mettre à jour et désinstaller, et les fichiers de configuration (fichiers Java Properties), dont il notifie les modifications auprès du service *Configuration Admin*. Bien que l'implémentation Pelix de *FileInstall* puisse surveiller et notifier des modifications de n'importe quel fichier, elle n'est utilisée que pour notifier *Configuration Admin* des événements concernant les fichiers de configuration.

3.6 Comparaison avec les modèles existants

Dans cette section, nous présentons une comparaison entre Pelix, iPOPO et certains des modèles existants proches de leurs concepts : OSGi [OSG12a], iPOJO [Esc08], Fractal [BCL+06] et SCA [SCA11]. Cette comparaison reprend les critères proposés par CRNKOVIC dans [CSVC11].

3.6.1 Cycle de développement d'un composant

Modèle de composant	Modélisation	Implémentation	Empaquetage	Déploiement
Fractal	Langage ADL (Fractal ADL, Fractal IDL), Annotations (Fraclet)	Java (Julia, AOKell), C/C++ (Think), .NET (FracNet)	Dossier de fichiers	À l'exécution
OSGi	N/A	Java	Archive Java (JAR)	À la compilation et à l'exécution
Pelix	N/A	Python	Module Python	À l'exécution
iPOJO	Annotations, XML	Java	Bundle OSGi	À l'exécution
iPOPO	Décorateurs	Python	Bundle Pelix	À l'exécution
SCA	SCA Composite	Multi-langage	Selon implémentation	Selon implémentation

TABLE 3.2 – Classification selon le cycle de vie des composants
Correspond à la table 1 dans [CSVC11].

La table 3.2 présente la gestion du cycle de vie d'un composant, telle que définie par [CSVC11] pour différents modèles à composants. Le cycle de vie d'un composant correspond ici à son cycle de développement, dont [CSVC11] retient 5 étapes :

1. Le composant est un ensemble d'exigences ;

13. `http://felix.apache.org/documentation/subprojects/apache-felix-file-install.html`

2. Phase de modélisation : le composant est un ensemble de modèles ;
3. Phase d'implémentation : le composant est un code source, associé à des méta-données ;
4. Après le déploiement, le composant est intégré dans un environnement d'exécution ;
5. À l'exécution le composant est un binaire (*object-code*) utilisable par la plate-forme cible.

Un modèle de composant peut ainsi assister la réalisation de certaines étapes de ce cycle de vie. Durant la phase de modélisation, des annotations dédiées ou des approches prédéfinies peuvent être préconisées par le modèle. Le modèle peut obliger à utiliser des éléments de construction pour l'implémentation d'un composant dans un langage particulier, *e.g.* les interfaces en Java, ou encore spécifier des règles de nommage. Enfin, il est possible que le modèle prenne en charge la transformation menant d'une étape à l'autre du cycle de vie, par exemple en générant du code source (*stub*) à partir des interfaces spécifiées durant la phase de modélisation.

La classification proposée par [CSVC11] retient quatre critères afin de caractériser le support de ces différentes étapes par les modèles à composants :

— Modélisation : un modèle peut assister la description de composants et du système qui les hébergera. Les modèles peuvent par exemple se baser sur des langages de description d'architecture (ADL) pour définir les interactions entre composants.

— Implémentation : un modèle peut participer à la génération de code source voire de code binaire, notamment lorsqu'il ne dépend pas d'un langage particulier. Il peut également cibler un langage d'implémentation spécifique et nécessiter l'utilisation de certains de ses outils.

— Empaquetage : ce critère présente le format de stockage utilisé par un modèle pour charger le code et les métadonnées d'un composant.

— Déploiement : ce critère indique à quel moment un composant est déployé dans son environnement d'exécution. Un composant peut tout d'abord être intégré durant l'étape de compilation. Par exemple, les références aux interfaces Java requises par un composant sont liées durant cette étape. Cette technique permet également, mais pas nécessairement, d'intégrer les composants et leur plate-forme en une seule image exécutable. Un composant est généralement déployé à l'exécution, que ce soit par ajout ou mise à jour : sa plate-forme ou son conteneur doivent alors lui permettre d'être lié au reste de l'application.

3.6.2 Liaisons entre composants

La table 3.3 montre les différentes définitions des liaisons entre composants des modèles présentés.

Le type d'interface indique si les modèles à composants s'appuient généralement sur des interfaces définissant un ensemble d'opérations (méthodes) ou un ensemble de ports de données et d'événements. Dans le premier cas, un composant en invoque un autre de la même manière que dans l'approche objet. Dans le second cas, un composant transmet des données depuis l'un de ses

Modèle de composant	**Type d'interface**	**Distinction *Provides/Requires***	**Fonctionnalités particulières**	**Langage d'interface**	**Niveaux d'interface**
Fractal	Opération	Oui	Interface composant et interface de contrôle	IDL, Fractal ADL, Java ou C, protocole comportemental	Syntaxique, Comportemental
OSGi	Opération	Oui	Interfaces dynamiques	Java	Syntaxique
Pelix	Opération	Oui	Interfaces dynamiques et modifiables, *Duck Typing*	Python (*Duck-Typing*)	Syntaxique
iPOJO	Opération	Oui	Injection de dépendance	Java	Syntaxique
iPOPO	Opération	Oui	Injection de dépendance	Python (*Duck-Typing*)	Syntaxique
SCA	Opération	Oui	Modèle extensible	Java, C, C++, WSDL	Syntaxique

TABLE 3.3 – Classification selon la construction - Spécification des interfaces
Correspond à la table 2 dans [CSVC11].

ports de sortie vers le port d'entrée d'un autre ; le traitement sera ensuite effectué par le second composant sur un événement (arrivée des données, horloge, ...).

La plupart des modèles différencient les interfaces que les composants fournissent (*Provides*) à leur environnement de celles qu'ils requièrent (*Requires*). Ceci permet notamment de définir explicitement les dépendances d'un composant et simplifie ainsi la réutilisation d'un composant dans de nouvelles architectures.

Le langage d'interface indique si une interface peut être définie dans le langage d'implémentation que cible le modèle ou à l'aide d'un langage de description d'interface (IDL). L'utilisation d'un IDL permet de décrire une interface qui pourra être implémentée dans différents langages.

Enfin, [CSVC11] retient trois niveaux de définition d'interfaces, extraits de [BJPW99] :

— Syntaxique : ce niveau vérifie que les composants liés utilisent les mêmes types de données et les mêmes signatures de méthodes. Cette vérification peut être effectuée durant la liaison (liaison C ou Java) ou à l'exécution (*duck-typing* [14] de Python).

— Sémantique fonctionnelle : ce niveau renforce le précédent en ajoutant les définitions de pré- et post-conditions aux méthodes ainsi que d'invariants.

— Comportemental : ce niveau permet d'exprimer des contraintes sur le comportement du composant appelé, allant de contraintes temporelles à celles ne l'autorisant à n'entrer que dans certains états.

14. James Whitcomb Riley : "*When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.*"

Modèle de composant	Liaison exogène	Liaison verticale	Styles d'interactions	Type de communication
Fractal	Oui	Délégation, Agrégation	Multiples	Synchrone, Asynchrone
OSGi	Non	Non	Requête–Réponse, Événementielles	Synchrone
Pelix	Non	Non	Requête–Réponse, Événementielles	Synchrone
iPOJO	Oui	Délégation, Agrégation	Requête–Réponse [15]	Synchrone
iPOPO	Oui	Non	Requête–Réponse [15]	Synchrone
SCA	Oui	Délégation, Agrégation	Requête–Réponse, Bidirectionnelles	Synchrone, Asynchrone

TABLE 3.4 – Classification selon la construction - Liaisons et interactions
Correspond à la table 3 dans [CSVC11].

La table 3.4 présente les méthodes de liaison et d'interaction utilisées par les différents modèles à composants présentés.

Une liaison exogène signifie que la logique de liaison est traitée en dehors du composant, à l'aide de connecteurs (adaptateur, *proxy*, *broker*, ...). Son opposée est la liaison endogène, traitée sans intermédiaire selon un protocole interne du composant.

La liaison verticale est la capacité d'un modèle à représenter un ensemble de composants comme un seul composant, généralement appelé composite. Si ce composite fournit tous les services de ses composants, il s'agit d'une agrégation ; s'il n'en fournit qu'une partie, il s'agit d'une délégation.

Le style d'interaction précise comment les dialogues entre composants s'effectuent. Le mode requête-réponse est le plus utilisé : le composant consommateur émet une requête et attend une réponse (un résultat) de la part du composant cible. Les appels de méthodes font partie de cette catégorie. Dans un mode événementiel, le composant consommateur émet un événement pour déclencher un traitement dans un composant cible. Des données peuvent être associées directement à cet événement, comme dans le modèle *publish/subscribe*, ou l'événement peut démarrer le traitement de données placées dans une file prédéfinie, *e.g.* dans le port d'entrée d'un composant d'un modèle *port-based*. D'autres styles d'interaction sont disponibles. Nous citerons par exemple le modèle *pipe & filter*, aussi appelé *pipeline* ou tube, dans lequel des données sont placées en entrée d'une chaîne de composants. Ces composants vont chacun leur tour traiter et filtrer les données résultant du traitement de leur(s) prédécesseur(s).

Enfin, les modèles de composants supportent généralement des appels de méthode synchrones, c'est-à-dire pour lesquels le composant attend le résultat de son appel avant de continuer son

15. Les interactions événementielles sont réalisables à travers le service `EventAdmin` fourni par la plate-forme sous-jacente (OSGi ou Pelix).

exécution. Ils peuvent également supporter des appels asynchrones, pour lesquels le composant continue son exécution, parallèlement au traitement de son appel par le composant cible.

3.6.3 Propriétés extra-fonctionnelles

Enfin, la table 3.5 montre comment chaque modèle à composants gère les propriétés extra-fonctionnelles (PEF).

Modèle de composant	Gestion des PEF	Spécification des PEF	Composabilité des PEF
Fractal	Exogène par collaboration	Capacité à ajouter des propriétés (par ajout de contrôleurs de propriété)	N/A
OSGi	Endogène par collaboration	Limitée aux services (*hooks* de R5)	N/A
Pelix	Endogène par collaboration	N/A	N/A
iPOJO	Exogène par collaboration	Propriétés définies et gérées par des gestionnaires (*handlers*)	N/A
iPOPO	Exogène par collaboration	Propriétés définies et gérées par des gestionnaires (*handlers*)	N/A
SCA	Endogène par collaboration	N/A	N/A

TABLE 3.5 – Classification selon le traitement des Propriétés Extra-Fonctionnelles
Correspond à la table 4 dans [CSVC11].

La gestion des propriétés extra-fonctionnelles peut être exogène, c'est-à-dire gérée hors composant, ou endogène, gérée par le composant lui même. Elle peut également être effectuée globalement (*system-wide*), *i.e.* toutes ces propriétés sont gérées par le modèle à composants ou par la plate-forme sous-jacente, afin d'atteindre des objectifs globaux (qualité de service, ...), ou bien par collaboration, auquel cas chaque composant est géré individuellement et collabore avec ses composants voisins pour atteindre des objectifs individuels, de groupe ou globaux.

Certains modèles permettent de spécifier les propriétés extra-fonctionnelles, c'est-à-dire de définir celles qui devront être associées aux composants. À l'inverse, d'autres modèles peuvent ne pas accepter de telles propriétés ou les autoriser sans pour autant avoir connaissance de leur présence ni de leurs actions.

Enfin, les propriétés extra-fonctionnelles peuvent être mises en relation lors du regroupement des composants qui leurs sont associés. Cette composabilité est généralement le résultat de la liaison verticale de composants, mais est plus complexe à gérer et est, de ce fait, peu souvent mise en œuvre par les modèles à composants. Il s'agit en effet de fusionner des propriétés ayant généralement des sémantiques importantes, telles que la qualité de service, mais qui ne peuvent que rarement l'être de manière automatisée.

3.7 Conclusion

Ce chapitre a présenté Pelix, une plate-forme à services orientés objet pour Python reprenant les concepts de la spécification OSGi. Pelix introduit la notion de *bundle* Python en ajoutant un cycle de vie aux modules de ce langage. Cette plate-forme apporte également un registre de services et la gestion des événements qui lui sont associés. Enfin, Pelix fournit un certain nombre de services de base, tels qu'un *shell*, un serveur HTTP et quelques services suivant plus ou moins la spécification OSGi, tels que *Event Admin* et *Configuration Admin*.

Nous avons également décrit iPOPO, un modèle à composants orientés service pour Python, inspiré d'iPOJO. Les composants de ce modèle sont des instances de classes Python manipulées à l'aide de décorateurs. Chaque composant est géré par son conteneur qui lui-même s'appuie sur des *handlers* afin de traiter les différents aspects du composant (dépendances, services fournis, propriétés, ...).

Pelix et iPOPO ont ensuite été comparés à leurs modèles respectifs à l'aide des critères de classification spécifiés dans [CSVC11]. Leurs caractéristiques apparaissent très proches de celles d'OSGi et d'iPOJO, en prenant en compte les spécificités de Python face à celles de Java.

Le projet Pelix/iPOPO est le socle de la plate-forme Cohorte. iPOPO a permis de développer très rapidement les composants Python de cette plate-forme. Sa proximité de contexte et d'API avec iPOJO a grandement accéléré le développement de leur pendant Java. Les services fournis par Pelix ont également simplifié le développement de Cohorte (services distants, ...) mais aussi la phase de débogage (commandes *shell* spécialisées, ...).

Pelix/iPOPO	**LOC**
Total plates-formes Pelix/iPOPO	10 504
dont Pelix	1 484
dont iPOPO	2 599
core	*1 084*
handlers	*937*
décorateurs (manipulation)	*465*
dont Services Distants	2 290
core	*805*
Tests Unitaires (comptés indépendamment)	7 640

TABLE 3.6 – Lignes de code d'iPOPO

La table 3.6 présente le nombre de lignes de code (LOC) nécessaires au développement de Pelix et iPOPO. La première ligne du tableau indique le nombre de lignes de code que compte le projet Pelix/iPOPO (hors tests) ; les lignes suivantes détaillent certaines parties du code et la dernière ligne les tests unitaires. Les tests unitaires couvrent 78% du code de Pelix/iPOPO. Seules les lignes contenant des instructions sont comptées ; les commentaires, la documentation et les

lignes vides sont omis. Le nombre de lignes de code a été mesuré à l'aide de l'utilitaire *cloc.pl* [16].

Le projet Pelix/iPOPO est un projet libre, diffusé sous les termes de la licence Apache Software License 2.0 [17] (ASL 2.0). Il est inscrit sur PyPI [18] (*Python Package Index*), l'index central des paquets Python. Son code source est disponible sur GitHub [19].

16. `http://cloc.sourceforge.net/`
17. `http://www.apache.org/licenses/LICENSE-2.0`
18. `https://pypi.python.org/pypi/iPOPO`
19. `https://github.com/tcalmant/ipopo`

CHAPITRE 4

CONTRIBUTION 2 : COHORTE HERALD

> *Le herault poulsant sa voix plus que devant, s'escria si hault, que son cry fut ouy de toute l'assemblée.*

Jacques AMYOT, *Les Vies des Hommes Illustres de Plutarque*, 1818

Sommaire

4.1 Introduction 67
4.2 Modèle 68
4.2.1 Concepts 68
4.2.2 Format de message 69
4.2.3 Structure 70
4.3 Découverte de pairs 71
4.4 Émission de messages 72
4.4.1 Modes de transmission 72
4.4.2 Implémentation 74
4.5 Services distants 74
4.5.1 Découverte de services 75
4.5.2 Messages d'appels distants 77
4.6 Shell distant 78
4.7 Conclusion 78

4.1 Introduction

Cohorte Herald [1] est le sous-système de Cohorte assurant les échanges entre les isolats d'une même application, par transmission de messages. Il est en charge de la découverte de chacun de ces isolats, de la transmission des messages internes de la plate-forme, mais également de la prise en charge de l'import et l'export des services distants de chaque isolat Cohorte. L'objectif ayant mené à son développement est d'obtenir la capacité à émettre un message vers un ou plusieurs

1. *Herald* est le terme anglais pour héraut, officier chargé de la transmission et de l'annonce de messages importants à l'époque médiévale

isolats sans avoir à connaître le protocole de communication sous-jacent, ni à prendre en compte ses contraintes.

Bien que mis au point pour les besoins de Cohorte, Herald est utilisable par n'importe quelle application OSGi. Cette bibliothèque a été développée en Python et en Java, en utilisant une approche à composants orientés service, afin de la rendre particulièrement dynamique. Il est par exemple possible d'ajouter, mettre à jour et supprimer des fournisseurs de transport en cours d'exécution.

4.2 Modèle

4.2.1 Concepts

Dans Herald, la communication s'effectue à l'aide de messages. Chaque message est distingué par un identifiant unique et est décrit par un sujet, associé à un contenu. L'identifiant, aussi appelé UID (*Unique IDentifier*), garantit qu'un message est traité une et une seule fois. Le sujet est une chaîne de caractères lisibles quelconque, généralement respectant la syntaxe URI (*Uniform Resource Identifier*). Les composants qui souhaitent être notifiés de la réception d'un message doivent s'abonner au sujet du message.

Ces messages sont transmis d'un pair émetteur à un pair ou à un groupe de pairs cibles. Un pair est la représentation d'un isolat Cohorte ou d'une application utilisant Herald. Chaque pair est distingué par un identifiant unique et porte un nom reconnaissable. Ce dernier peut être partagé par plusieurs pairs, notamment dans le cas d'isolats dupliqués. De la même manière, il est possible d'associer à chaque pair l'identifiant unique et le nom du nœud sur lequel il est exécuté (généralement le nom de la machine hébergeant le pair). Ces attributs sont soit déterminés à partir de la configuration de Cohorte, soit indiqués par l'opérateur de l'application, soit copiés de l'identifiant et du nom du pair. Le nom de nœud permet de déterminer les pairs s'exécutant sur une même machine, mais aussi de communiquer avec eux spécifiquement.

Un message peut être émis vers un groupe de pairs : chaque pair est associé à un certain nombre de groupes. Le nom d'un pair, l'identifiant et le nom de son nœud font partie des noms de groupes auxquels il appartient automatiquement. Il existe également un groupe spécial nommé « all »contenant la liste de tous les pairs connus de l'émetteur. Un pair peut appartenir à d'autres groupes, dont le nom et la sémantique dépendent de l'application. Enfin, la description d'un pair n'est valide que si elle contient la description d'au moins un *accès*. Un accès est l'association de l'identifiant d'un fournisseur de transport à la configuration nécessaire à ce dernier pour communiquer avec le pair décrit. Si deux pairs se connaissent un accès ayant le même identifiant de fournisseur de transport, alors ils pourront communiquer *via* ce canal.

4.2.2 Format de message

Comme nous l'avons mentionné, un message Herald possède un identifiant, un sujet et un contenu. De plus, chaque message est associé à des métadonnées, c'est-à-dire un ensemble de propriétés utilisées par l'émetteur pour décrire le message ou son contenu. Par exemple, si un émetteur utilise différents formats de sérialisation (JSON, BSON, XML, ...) pour un même sujet de message, il peut indiquer le format utilisé dans ces métadonnées. Enfin, un message est également associé à des en-têtes (*headers*), concernant l'émission du message et sa cible. Les en-têtes obligatoires sont l'identifiant du message et le numéro de version du format de message ; sans ces éléments, Herald ne traite pas le message. Il indique également, dans les en-têtes, la date de création du message, qui peut être différente de sa date d'émission. Ainsi, un abonné peut réordonner les messages provenant d'un même pair selon leur date de création, même s'ils ont été reçus voire émis dans un ordre différent. Tandis que les métadonnées sont réservées à la description du message par l'émetteur, les en-têtes peuvent être utilisés par les composants et les extensions de Herald (informations pour transports, ...).

Par défaut, les messages Herald sont transmis au format JSON et le contenu au format Jabsorb (format JSON contenant des informations de typage Java). Cependant, un service de transport peut utiliser n'importe quel format correspondant à ses exigences.

```
{
    "subject": "sensors/temperature/update",
    "headers": {
        "herald-version": 1,
        "sender-uid": "0ef4051d-b3c6-4301-a513-0c060e621829",
        "timestamp": 1435096156651,
        "uid": "44ECAC05076743849BBCCF1A2C8B9E0A",
        "herald-http-tansport-path": "/herald",
        "herald-http-tansport-port": 39171
    },
    "metadata": {},
    "content": {
        "javaClass": "java.util.HashMap",
        "map": {
            "unit": "Kelvin",
            "value": 300,
            "sensor_id": "7A60402981C2402B9BCADE609E9D4A63"
        }
    }
}
```

Listing 2 – Exemple de message Herald

Le listing 2 est un exemple de message transmis par Herald. Il a pour sujet `sensors-/temperature/update` et contient un dictionnaire décrivant la dernière température lue à

l'aide d'un capteur. Le contenu est sérialisé au format Jabsorb : l'indication `JavaClass` permet aux récepteurs Java de connaître le type d'objet à instancier lors du décodage du message. Étant donné qu'il s'agit du format utilisé par défaut par Herald, cette information n'est pas indiquée dans les métadonnées.

Les en-têtes sont initialement configurés par Herald pour indiquer l'identifiant unique du message, sa date de création et l'identifiant unique du pair émetteur. Ces informations permettent aux récepteurs d'ignorer le message s'ils l'ont déjà reçu ou s'ils ne gèrent pas cette version du format de message. L'identifiant de l'émetteur leur permet de déterminer à qui envoyer un message de réponse. Les deux autres en-têtes présents dans cet exemple ont été inscrits par le transport HTTP. Ils contiennent les informations nécessaires au service de transport HTTP du récepteur pour répondre à l'émetteur si ce dernier n'est pas inscrit dans l'annuaire, *e.g.* s'il n'a pas encore été découvert ou s'il n'appartient pas à la même application.

4.2.3 Structure

La figure 4.1 montre la structure générale de Herald, c'est-à-dire le cœur de Herald et un fournisseur de transport. Les ronds blancs représentent les services fournis par les composants, selon une interface publique ; les ronds gris représentent des services dépendants d'une implémentation spécifique et ne seront donc pas décrits.

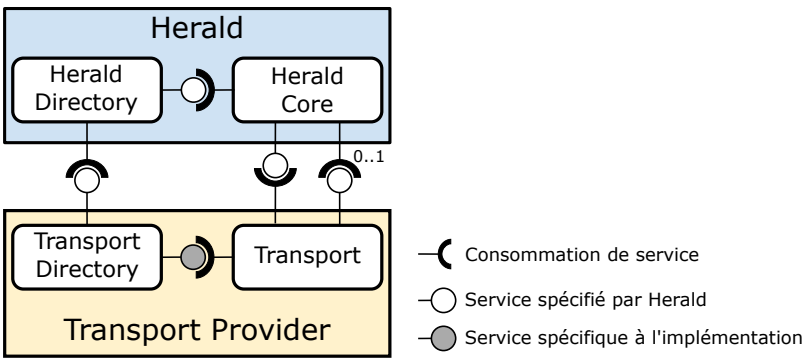

FIGURE 4.1 – Vision globale des composants de Herald

La boucle de consommation de service est possible car le service fournit par *Herald Core* est toujours actif, et celui qu'il consomme est optionnel. Le premier service permet à un transport de notifier *Herald Core* de la réception d'un message ; le second d'utiliser un transport pour émettre un message.

Herald est constitué de deux principaux composants internes :

— *Herald Directory* conserve les descriptions des pairs découverts, tant qu'elles restent valides, *i.e.* tant que leur pair dispose d'au moins un accès valide.

— *Herald Core* est utilisé par les émetteurs afin d'envoyer leurs messages sans qu'ils aient à connaitre le protocole sous-jacent. Il permet aux récepteurs de s'abonner à la réception de messages selon leur sujet. Ce composant est lui-même notifié de la réception de messages par les fournisseurs de transport, et se charge d'en informer leurs abonnés.

Les fournisseurs de transport sont quant à eux des boites noires, enregistrant chacun deux services :

— un service annuaire spécifique, utilisé par *Herald Directory* pour construire la description d'un pair et pour transmettre celle du pair local.

— un service d'émission, se chargeant de l'envoi d'un message Herald selon le protocole sur lequel il s'appuie.

Quand il émet ou reçoit un message, un fournisseur de transport peut y ajouter des métadonnées, notamment pour indiquer comment renvoyer une réponse à ce message.

4.3 Découverte de pairs

Herald se charge de la découverte des pairs d'une même application. Dans le cas de Cohorte, cela signifie que les isolats se découvrent mutuellement à l'aide de cette bibliothèque, plutôt qu'en partageant un annuaire centralisé. De cette manière, un isolat ne peut être la cible de messages que si sa couche de communication est complètement et correctement chargée.

Chaque fournisseur de transport, s'il en a la capacité, doit se charger de découvrir les pairs à sa portée. Cette phase de découverte peut utiliser les mécanismes du protocole sur lequel le fournisseur de transport s'appuie (annuaire XMPP, etc.), mais aussi un protocole de découverte standard (mDNS, SSDP, etc.) ou encore un mécanisme spécifique à son implémentation.

Lorsqu'un pair `P` est détecté, le fournisseur de transport lui envoie un message ayant pour sujet `herald/discovery/step1`, contenant la description du pair local, noté `L`. Ce message marque le début de la synchronisation de la découverte de ces pairs, décrit par la figure 4.2.

1. Le message `step1` est émis par le pair local `L` ayant découvert `P`, contenant sa propre description.
2. Ce message est reçu par le fournisseur de transport de `P`. Celui-ci enregistre `L` dans *Herald Directory*, mais ne notifie ni Herald ni ses composants de cet ajout.
3. `P` émet un message `step2` vers `L`, contenant sa propre description.
4. Le fournisseur de transport de `L` reçoit ce message. Il enregistre la description de `P` dans *Herald Directory*, puis en notifie Herald et ses composants. À cet instant, les composants de `L` commencent à communiquer avec `P`, et ce dernier est capable de répondre à ces messages.
5. `L` envoie un message `step3` à `P` pour terminer la synchronisation.

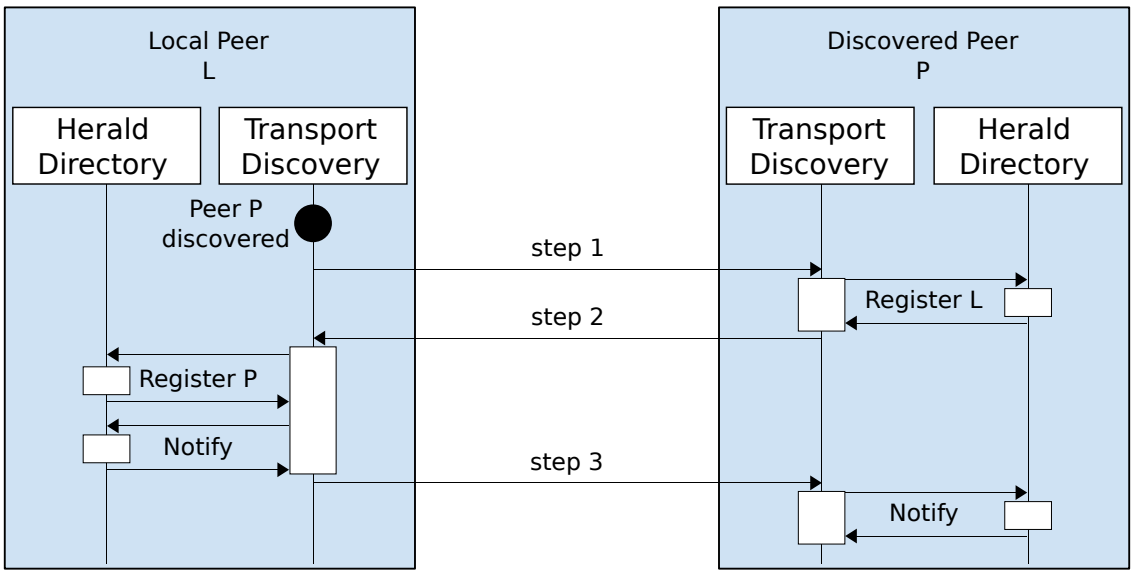

FIGURE 4.2 – Synchronisation des annuaires Herald

6. `P` reçoit ce message et notifie Herald et ses composants de la découverte de `L`. Les composants de `P` commencent à communiquer avec ceux de `L`.

Ce mécanisme de découverte en trois étapes est inspiré de la séquence de synchronisation du protocole TCP : SYN, SYN-ACK, ACK. Il permet de s'assurer que les deux pairs se connaissent avant d'autoriser la transmission de messages et évite ainsi qu'un émetteur puisse envoyer des requêtes à un pair qui ne pourra pas lui répondre. Dans le cas où ce mécanisme ne peut être mené à terme, *e.g.* si l'un des pairs disparait ou ne répond pas dans les délais, l'enregistrement des pairs est annulé.

4.4 Émission de messages

4.4.1 Modes de transmission

Herald supporte trois sémantiques d'émission de message : *fire*, *send* et *post*. Le mode *fire* indique que l'émetteur n'attend pas de réponse. Cependant, il est mis en attente jusqu'à obtention de l'accusé de réception du pair distant, d'une erreur de transmission ou de l'indication qu'aucun abonné n'écoute le message dans le pair distant. Un message émis en mode *fire* peut cibler un pair ou un groupe de pairs. *Send* est un mode requête-réponse synchrone : l'émetteur attend de recevoir un message de réponse avant de pouvoir continuer son exécution. Ce mode est implémenté par dessus le mode *fire* : l'émetteur envoie un message et reçoit un acquittement, puis attend de recevoir un nouveau message portant l'information qu'il répond au premier. Un message ne peut être émis dans ce mode que vers un pair : la synchronisation des réponses sur un groupe de pairs

serait complexe à gérer et potentiellement une source d'erreurs. *Post* est la version asynchrone du mode *send*. Celle-ci fonctionne sur le même principe, mais plutôt que de bloquer l'exécution de l'émetteur en attendant une réponse, ce dernier peut indiquer deux méthodes à appeler en cas de succès ou d'échec de la transmission du message. Comme *fire*, ce mode peut être utilisé pour envoyer un message à un pair ou à un groupe de pairs.

Il est également possible pour un pair de répondre à un message qu'il a reçu : ce nouveau message est alors envoyé en mode *fire* et une métadonnée supplémentaire indique l'identifiant du message auquel il répond. La réponse à un message s'effectue au niveau du fournisseur de transport : celui-ci peut se baser sur les informations qu'il a enregistrées lors de la réception du message original pour émettre sa réponse, notamment si le pair émetteur n'est pas encore inscrit dans le service annuaire de Herald.

La figure 4.3 décrit la transmission d'un message avec Herald :

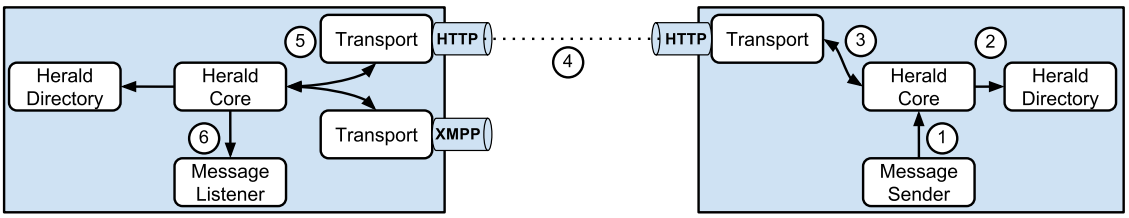

FIGURE 4.3 – Transmission de message Herald

1. L'émetteur crée un objet *Message*, en indiquant son sujet et son contenu. L'identifiant et l'horodatage du message sont remplis automatiquement. L'émetteur transmet au service *Herald Core* le message et sa cible (pair ou groupe de pairs).
2. *Herald Core* va ensuite demander à son annuaire, *Herald Directory*, les informations sur le ou les pairs indiqués. De cette manière, Herald normalise la cible donnée par l'émetteur (identifiant ou nom de pair, objet représentant un pair, etc.) en un objet représentant le pair. Si le pair est inconnu, le service renvoie une erreur à l'émetteur.
3. Le service *Herald Core* essaye alors d'envoyer le message par chaque transport disponible et pouvant communiquer avec la cible, jusqu'à ce que l'émission réussisse.
4. Le fournisseur de transport convertit le message Herald selon son protocole et l'émet.
5. Le transport de la cible reçoit le message, le convertit selon le modèle Herald, et y ajoute des informations sur l'émetteur, dépendantes de son implémentation. Ensuite, le message est transmis au service *Herald Core*.
6. Ce dernier cherche alors la liste des abonnés inscrits au sujet du message. Si aucun abonné n'est trouvé, un message d'erreur *no-listener* est renvoyé à l'émetteur. Dans le cas contraire, le lien entre l'émetteur et le récepteur est libéré. Par exemple, si le message a été reçu à

travers une requête HTTP, cette libération se traduit par une réponse HTTP 200 OK, ce qui permet à l'émetteur de réutiliser la connexion actuelle. S'il s'agit d'un message XMPP, cette libération est implicite : l'émetteur n'est pas en attente active d'une réponse ou d'un acquittement.
Enfin, les abonnés sont notifiés de l'arrivée du message.

4.4.2 Implémentation

Notre implémentation de Cohorte Herald supporte actuellement les transports HTTP et XMPP.

Le protocole HTTP est utilisé en mode requête-requête, c'est-à-dire que toute réponse d'un récepteur est émise comme une nouvelle requête vers l'émetteur. Ce mécanisme permet de passer outre les délais de réponse imposés par le protocole HTTP (30 secondes). Les pairs communiquant à travers HTTP sont découverts à l'aide d'un protocole *fait maison*, basé sur des paquets UDP multicast. Certains de ces paquets sont émis à intervalles réguliers par chaque pair afin d'indiquer et de confirmer sa présence (*heart beat*). Les pairs peuvent soit être configurés afin de rejoindre un groupe (*i.e.* une adresse) et un port multicast particulier ou afin d'écouter un port HTTP spécifique, soit utiliser la configuration par défaut.

Concernant XMPP, un message Herald est transmis comme n'importe quel message instantané selon ce protocole. Lorsque XMPP est utilisé pour le transport de messages, la découverte de pairs se fait également par ce protocole, à travers les événements de présence émis par le serveur XMPP auquel les pairs sont tous connectés. Les pairs doivent alors être configurés pour connaître le serveur sur lequel ils doivent se connecter. L'opérateur peut indiquer des informations d'authentification ou laisser les pairs se connecter au serveur anonymement.

Cohorte Herald peut être étendu par de nouveaux protocoles de découverte et de transport, et ce même à l'exécution. Par exemple, des composants sont actuellement développés afin de découvrir des pairs et de leur permettre de dialoguer à travers une liaison Bluetooth RFCOMM, *via* un protocole binaire spécifique.

Dans tous les cas, le contenu émis à travers le protocole de transport contient l'intégralité des informations du message, dans un format proche de celui présenté dans le listing 2. Les implémentations des transports HTTP et XMPP transmettent ces données au format JSON, directement enveloppées dans le corps de leur requête HTTP ou message XMPP.

4.5 Services distants

Cohorte Herald fournit également un transport pour Pelix et Cohorte Remote Services, basé uniquement sur les messages. Les composants de cette fonctionnalité sont regroupés sous le nom

Herald-RPC. Ceux-ci ne sont que consommateurs de la partie principale de Herald, décrite précédemment.

Selon le modèle de Pelix Remote Services, Herald-RPC fournit trois services, chacun implémenté par un composant particulier.

Le composant de découverte, *Discovery*, a pour tâche d'émettre un message à chaque ajout (`add`), mise à jour (`update`) et suppression (`remove`) de service exporté. Il traite également ces messages lorsqu'ils sont reçus depuis d'autres pairs.

Le composant *Importer* se charge de créer un proxy (mandataire de service), simulant l'interface du service, et de l'enregistrer comme un service local, selon les informations transmises par le composant *Discovery*. Quand un consommateur utilise ce proxy, *Importer* convertit les paramètres d'appel en un message RPC [2] et le diffuse vers le pair exportant le service concerné.

Enfin, le composant *Exporter* analyse les messages RPC reçus, convertit chacun d'entre eux en un appel de méthode et renvoie son résultat ou une éventuelle erreur dans un message de réponse.

4.5.1 Découverte de services

La découverte des services fournis par les autres pairs d'une application dépend directement du mécanisme de découverte de pairs par Herald. Lorsque le composant *Herald Directory* enregistre un nouveau pair, *Discovery* est notifié afin de lui envoyer un message `contact`, comme le montre la figure 4.4. Ce message contient la liste de tous les services exportés par le pair local. Le nouveau pair enregistre ces derniers et répond alors avec un message `add` contenant la liste de tous les services qu'il exporte. L'utilisation du message `add` assure l'emploi du même traitement que lors de l'export d'un nouveau service par un pair.

Les messages de découverte sont décrits dans le tableau 4.1.

Sujet	Contenu	Description
`contact`	Description complète des services exportés par l'émetteur	Utilisé lors de la découverte d'un pair. Le pair récepteur peut émettre un message `add` en réponse.
`add`	Description complète des nouveaux services	Émis après la création des nouveaux *endpoints*.
`update`	Identifiant du *endpoint* et nouvelles propriétés	Émis dès que les propriétés du service exporté ont été modifiées.
`remove`	Identifiant du *endpoint*	Émis lorsqu'un *endpoint* est supprimé ou que le service est désinscrit.

TABLE 4.1 – Messages de découvertes

2. RPC : Remote Procedure Call

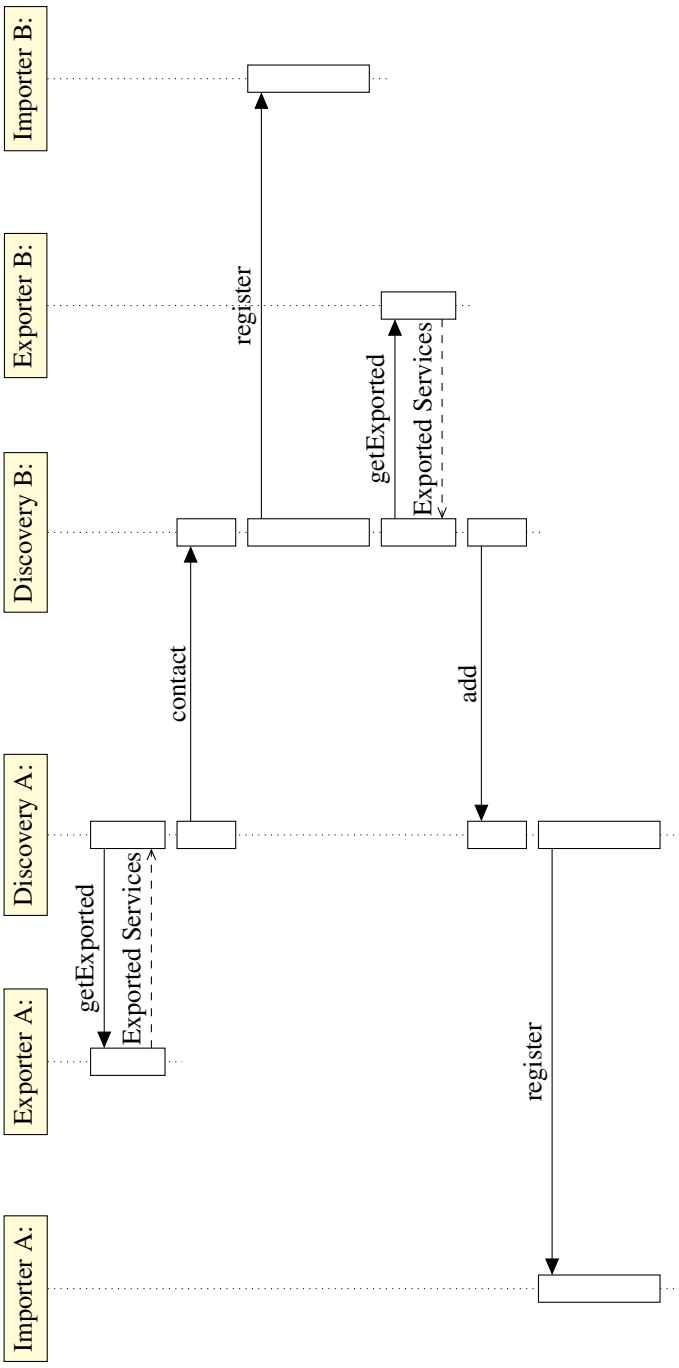

FIGURE 4.4 – Échange de services entre pairs

La description complète d'un service exporté, telle que citée dans le tableau 4.1, contient toutes les informations nécessaires pour construire un proxy, à savoir :

— l'identifiant du *endpoint*, généré par le composant *Exporter* ;

— les propriétés du service, telles que spécifiées par *OSGi Remote Service Admin*.

Le pair émetteur étant connu grâce aux méta-données du message Herald utilisé pour le transport, l'utilisation de Herald pour transmettre les messages RPC évite d'avoir à décrire plus précisément les méthodes d'accès aux *endpoints*.

4.5.2 Messages d'appels distants

Les messages transmis lors des appels distants s'inspirent du standard JSON-RPC 2.0 [JSO13]. La représentation des arguments et des valeurs de retour est modifiée afin d'y ajouter des informations de typage au format utilisé par la bibliothèque Jabsorb-ng [3]. Cette dernière permet de reconstruire des objets Java à partir des informations supplémentaires ajoutées à l'objet JSON de requête ou de réponse, sans avoir à modifier ni annoter les classes Java concernées. Ainsi, le contenu des messages Herald utilisés lors d'appels distants est de la forme décrite par les listings 3 et 4. Ces listings montrent le contenu du champ `content` dans des messages émis selon le format présenté dans le listing 2 (page 69).

```
{
  "javaClass": "java.util.HashMap",
  "map": {
     "id": "0f54155d-4160-4c11-915c-93baa1c81efa",
     "jsonrpc": "2.0",
     "method": "endpoint_10.add",
     "params": [2, 40]
  }
}
```

Listing 3 – Contenu d'un message d'appel distant sur un service

```
{
  "javaClass": "java.util.HashMap",
  "map": {
     "id": "0f54155d-4160-4c11-915c-93baa1c81efa",
     "jsonrpc": "2.0",
     "result": 42
  }
}
```

Listing 4 – Contenu d'une réponse d'un service à un appel distant

3. https://github.com/isandlaTech/cohorte-org.jabsorb.ng/

4.6 Shell distant

Herald fournit également une interface au shell Pelix, présenté dans la section 3.5.1. Celle-ci permet d'accéder au shell d'un pair en ne connaissant que son identifiant unique : il n'est pas nécessaire pour le client d'en connaître les modalités d'accès. Cette interface est elle-même démarrée à l'aide d'une commande du shell Pelix du pair client et peut donc être lancée depuis n'importe quelle autre interface (console texte, shell distant, ...).

Cette implémentation fournit un *I/O Handler* basé sur les messages : chaque ligne de commande est envoyée par un message en mode *send* et attend un résultat de la part du pair ciblé. Ce dernier peut renvoyer des messages en mode *fire* pour indiquer des lignes à afficher au cours de l'exécution de la commande, mais aussi en mode *send*, notamment pour demander une entrée de la part de l'utilisateur (*prompt*).

Du fait de l'utilisation de messages Herald, ce shell bénéficie des tous les outils fournis par Herald, notamment le routage. Il devient ainsi possible d'accéder au shell d'un pair qui n'est pas directement accessible, et ce de manière transparente.

4.7 Conclusion

Ce chapitre a présenté Cohorte Herald, le canevas de communication de la plate-forme Cohorte. Celui-ci permet à des pairs de s'échanger des messages, tout en masquant les mécanismes de découverte et de transport utilisés. Cohorte Herald fournit une implémentation des Pelix Remote Services, afin qu'un pair puisse découvrir et consommer des services d'un autre pair, sans configuration particulière. Un shell distant a également été implémenté pour Pelix : tout pair peut accéder à distance au shell de n'importe quel autre pair hébergeant un agent faisant le pont entre les messages Herald et son shell local.

Deux protocoles de transport sont actuellement supportés : HTTP et XMPP. Le protocole HTTP est associé à un mécanisme de découverte de pairs s'appuyant sur des paquets UDP multicast et cible donc les réseaux locaux. Le protocole XMPP utilise un serveur central, pouvant être répliqué, et est généralement utilisé pour lier des pairs géographiquement éloignés ou faisant partie de réseaux différents.

Actuellement, deux pairs Herald ne peuvent communiquer que s'ils se sont découverts et s'ils disposent d'un protocole de transport commun. Une évolution du canevas est en cours de développement pour éliminer cette contrainte : le routage de messages. Avec ce nouvel outil, deux pairs ayant des transports différents ou ne pouvant se découvrir directement pourront transmettre leurs messages à travers un ou plusieurs pairs intermédiaires, aussi appelés pairs passerelles.

Une autre évolution en cours de développement est la création de tunnels TCP et UDP. Le concept d'un tunnel Herald est proche de celui d'un tunnel SSH. Un pair peut créer un serveur écoutant sur un port donné par l'opérateur et transmettre tous les paquets reçus vers un pair de

sortie. Ce dernier aura été configuré pour se connecter et transmettre les paquets à un serveur tiers, accessible depuis son réseau.

Cohorte Herald a été principalement développé en Python. Une version Java est également disponible, mais ne fournit pas toutes les fonctionnalités annexes (shell distant, ...). La table 4.2 présente le nombre de lignes de code (LOC) Python et Java nécessaires au développement de Cohorte Herald. Seules les lignes contenant des instructions sont comptées ; les commentaires, la documentation et les lignes vides sont omis. Le nombre de lignes de code a été mesuré à l'aide de l'utilitaire *cloc.pl* [4].

Cohorte Herald	**LOC Python**	**LOC Java**	**Python+Java**
Total Cohorte Herald	3 751	5 321	9 072
Noyau	1 165	2 213	3 378
Transports	1 453	2 320	3 773
HTTP	*734*	*1 455*	*2 189*
XMPP	*659*	*865*	*1 524*
Services Distants	445	595	1 040
Découverte	*90*	*213*	*303*
Jabsorb-RPC	*114*	*374*	*488*
Shell distant	277	0	277

TABLE 4.2 – Lignes de code de Cohorte Herald

Étant donné qu'il fait partie de Cohorte, le projet Cohorte Herald est un projet libre diffusé sous les termes de la licence Apache Software License 2.0 [5] (ASL 2.0). Son code source est disponible sur la forge en ligne GitHub [6]. La version Java de Cohorte Herald dépend du projet Cohorte Remote Services [7], une implémentation en Java des services distants de Pelix diffusée également sous licence ASL 2.0. Bien que la partie Python du projet soit inscrite sur PyPI [8] (*Python Package Index*), l'annuaire et dépôt central principal des paquets Python, la partie Java n'est pas inscrite sur Maven Central, son équivalent pour les binaires Java utilisables avec Maven.

4. `http://cloc.sourceforge.net/`
5. `http://www.apache.org/licenses/LICENSE-2.0`
6. `https://github.com/isandlaTech/cohorte-herald`
7. `https://github.com/isandlaTech/cohorte-remote-services`
8. `https://pypi.python.org/pypi/Cohorte-Herald`

CHAPITRE 5

CONTRIBUTION 3 : COHORTE RUNTIME

« Diviser pour mieux régner. »

Proverbe

Sommaire

5.1 Introduction 81
5.2 Modèle 82
5.3 Fichier de configuration d'un isolat 82
5.4 Architecture 84
5.5 Cycle de vie d'un isolat 86
5.5.1 Démarrage d'un isolat par *bootstrap* 86
5.5.2 Démarrage d'un isolat *exe* 88
5.5.3 Disparition d'un isolat 88
5.5.4 Arrêt de la plate-forme 89
5.6 Conclusion 90

5.1 Introduction

Ce chapitre présente Cohorte Runtime, la plate-forme d'exécution prenant en charge les isolats hébergés dans un même nœud. Elle a pour fonction de démarrer ces isolats à partir d'une configuration préparée par l'opérateur de l'application, de les surveiller et de les redémarrer en cas d'arrêt impromptu, *e.g.* après un crash d'isolat ou un arrêt provoqué par l'utilisateur.

Cohorte Runtime est capable de gérer des isolats contenant des artefacts (*bundles*, ...) et des composants : les composants sont eux-mêmes conditionnés dans des artefacts. Le démarrage et la configuration des artefacts et des composants sont totalement pris en charge par la plate-forme, à travers un *bootstrap* (amorce). Cohorte Herald assure la communications entre ces isolats, qu'il voit comme des pairs. Cohorte Runtime est également capable de traiter comme isolats des fichiers exécutables (fichiers *.exe*, ...) : la plate-forme démarre ces exécutables en ne configurant que leurs variables d'environnement puis surveille leur présence. En cas d'arrêt le fichier est à nouveau exécuté. Par la suite, nous appellerons les isolats de ce type des isolats *exe*, par opposition aux isolats démarrés à l'aide d'un *bootstrap*. Ces derniers sont simplement appelés isolats, possiblement en indiquant le langage d'implémentation de leurs composants.

Dans ce chapitre, nous présentons le modèle utilisé par Cohorte Runtime pour représenter un isolat et son contenu. Nous exposons également comment un opérateur peut configurer un isolat et enfin comment cette plate-forme gère et démarre un isolat.

5.2 Modèle

L'unité de base de Cohorte Runtime est le composant, tel que défini dans les modèles à composants orientés service et présenté section 2.4 (page 18 de ce manuscrit). Dans ce modèle, le composant est décrit par une fabrique (*factory*) permettant de l'instancier, par un nom d'instance et par des propriétés de configuration. La plate-forme se chargera de lui associer l'artefact qui fournit sa fabrique, soit par recherche, soit par configuration ; et, à partir de cet artefact, de déterminer son langage d'implémentation.

Un artefact est une unité de déploiement identifiée par un nom et un numéro de version. Il peut dépendre d'autres artefacts ; la plate-forme doit s'assurer que ses dépendances soient satisfaites avant de pouvoir exécuter l'isolat l'hébergeant. La plate-forme se charge d'associer chaque artefact au fichier qui l'empaquète, soit par recherche, soit par configuration.

Enfin, l'unité de travail de la plate-forme est l'isolat, formé d'un ensemble d'artefacts et de composants.

Cohorte Runtime se charge de compléter la liste des artefacts à intégrer à un isolat afin que ceux qui fournissent les fabriques des composants soient présents, ainsi que leurs dépendances. Un isolat porte un nom, permettant de l'identifier, et un type (*kind*) indiquant s'il s'agit d'un isolat *exe* ou précisant le langage d'implémentation de ses composants. S'il s'agit d'un isolat *exe*, il est décrit par le chemin d'accès au fichier à exécuter ainsi que par les paramètres à lui fournir. Dans le cas d'un isolat démarré par *bootstrap*, il est possible de définir les propriétés de la plate-forme qui sera démarrée. Dans les deux cas, il est possible d'indiquer des variables d'environnement qui seront accessibles par l'isolat. Il est important de noter qu'un isolat ne peut contenir que des composants implémentés dans le même langage.

Pour chaque élément du modèle, la plate-forme est capable d'extrapoler certaines informations. Par exemple, le langage associé à un isolat est déterminé selon le langage d'implémentation de ses composants, qui est lui-même déterminé selon le type des artefacts fournissant les fabriques desdits composants.

5.3 Fichier de configuration d'un isolat

Un isolat peut être défini soit par une configuration dans un fichier, soit programmatiquement. La seconde option n'est utilisée que par Cohorte Composer, qui est décrite dans le chapitre suivant. La première option permet, quant à elle, que l'opérateur d'une application Cohorte décrive les isolats qu'il souhaite voir démarrés et gérés par la plate-forme Cohorte Runtime.

```
{
    "name": "isolate-cache",
    "kind": "java",
    "boot_args": [
        "-Xms64M",
        "-Xmx512M"
    ],
    "properties": {
        "osgi.shell.telnet.port": "6003"
    },
    "bundles": [ {
            "symbolicName": "org.apache.felix.shell"
        }, {
            "symbolicName": "org.apache.felix.shell.remote",
        }, {
            "from": "remote-services.js"
        }
    ],
    "composition": [ {
            "factory": "cache-handler-factory",
            "name": "cache-handler",
            "properties": {
                "max-age": 3600
            }
        }
    ]
}
```

Listing 5 – Exemple de configuration d'isolat

Cette configuration est effectuée à travers un fichier au format JSON. Le listing 5 présente un exemple de fichier de configuration. Nous retrouvons ici les champs *name* et *kind* indiquant respectivement le nom et le type de l'isolat. La valeur *java* indique qu'il s'agit d'un isolat démarré par *bootstrap* qui hébergera des composants développés en Java. Des paramètres de démarrage (`boot_args`) peuvent être spécifiés : il s'agit en fait des paramètres à donner à l'interpréteur Python ou à la machine virtuelle Java lors de son démarrage, afin de configurer finement le comportement de l'isolat. Dans cet exemple, nous configurons les quotas d'utilisation mémoire de la machine virtuelle Java, ce qui est impossible à effectuer une fois que celle-ci a démarré. Vient ensuite la définition des propriétés de l'isolat. Selon le type d'isolat à démarrer, elles seront configurées au niveau de la plate-forme d'exécution du modèle à composants, *e.g.* les propriétés des plates-formes Pelix ou OSGi, ou de la plate-forme d'exécution du langage, *e.g.* les propriétés système de la machine virtuelle Java. Il est possible d'indiquer la liste des artefacts que doit contenir l'isolat à l'aide du champ *bundle*. Cette liste est généralement utilisée pour indiquer des artefacts qui fournissent des services ou des fonctionnalités sans requérir l'instanciation de leurs composants. C'est notamment le cas des artefacts qui inscrivent des services *via* l'API bas-niveau de la plate-forme OSGi, ou de ceux qui instancient eux-mêmes leurs composants. Enfin, il est pos-

sible de configurer des instances de composants, en indiquant pour chacun l'identifiant de leur fabrique, leur nom d'instance et leurs propriétés de configuration initiale. L'artefact fournissant cette fabrique et ses propres dépendances seront automatiquement ajoutés à la liste des artefacts à installer dans l'isolat.

L'analyseur de configuration de Cohorte Runtime permet l'import du contenu d'autres fichiers JSON, et leur fusion, avec ou sans surcharge, avec le contenu du fichier importateur. Le champ *from*, visible à la ligne 16 du listing 5, est ainsi remplacé à la volée par le contenu du fichier `remote-services.js`. Ce mécanisme permet notamment la définition de groupes de fonctionnalités, réutilisables à travers les différents isolats de l'application, ainsi que la centralisation de certaines configurations. Par exemple, la configuration de Cohorte Herald peut être partagée entre tous les isolats afin d'assurer qu'ils utilisent les mêmes protocoles de découverte.

5.4 Architecture

La plate-forme Cohorte Runtime s'appuie sur un isolat particulier : le moniteur (aussi appelé *forker*). Cet isolat est démarré à l'aide du *bootstrap* à la manière des autres isolats, avec une configuration qui lui est propre, correspondant au type d'isolat *forker*. Il s'agit du cœur de la plate-forme, se chargeant de gérer les isolats.

L'isolat moniteur est composé de deux grands éléments : le *Monitor*, qui utilise le modèle décrit précédemment pour gérer les isolats à haut niveau, et le *Forker*, qui a une vision des isolats plus proche du système hôte. Du fait de cette proximité, nous avons choisi de construire cet isolat à l'aide de composants développés en Python car ce langage permet d'avoir accès à des méthodes bas niveau spécifiques à chaque système d'exploitation. Le *Monitor* donne au *Forker* des représentations d'isolats pour que ce dernier les démarre. Le *Forker* s'appuie lui-même sur des services *Starter* pour démarrer les différents types d'isolats. Actuellement, il n'existe que deux *Starters*, l'un pour démarrer les isolats à l'aide du *bootstrap* et l'autre pour démarrer les isolats *exe*. La surveillance des isolats est gérée par un seul composant, l'*IsolateWatcher*. Lorsqu'une défaillance est détectée par ce composant, il notifie le *Monitor* afin que l'isolat fautif soit redémarré. Enfin, le composant *Status* se charge de conserver les machines à états finis représentant chaque isolat.

L'isolat moniteur héberge également un certain nombre de composants utilitaires. Les composants *repositories*, par exemple, analysent les fichiers présents dans les dépôts d'artefacts afin d'obtenir la liste de tous les artefacts disponibles, de déterminer leur langage d'implémentation, leurs dépendances ainsi que la liste des fabriques de composants qu'ils fournissent.

L'isolat moniteur peut démarrer des isolats par *bootstrap*. Le *bootstrap* est un script Python capable de récupérer la configuration de l'isolat auprès de l'isolat moniteur puis d'installer les artefacts et d'instancier les composants indiqués. Parmi ces composants se trouvent notamment ceux de Cohorte Herald, qui permettent au moniteur et au nouvel isolat de se découvrir et de

dialoguer. Chacun de ces isolats héberge un composant *Monitor Agent*, qui se charge d'indiquer la fin du chargement du nouvel isolat au composant *Monitor* de l'isolat moniteur à l'aide des services distants fournis par Herald. Ainsi l'isolat moniteur est notifié de la fin du chargement de l'isolat et de sa capacité à communiquer. Si cette notification n'est pas reçue par le moniteur dans un délai imparti (par défaut, 30 secondes), alors l'isolat est tué et la procédure de lancement recommence. Ce délai est évidemment configurable, étant donné qu'un isolat peut prendre plus de temps à démarrer sur des machines à ressources limitées ou aux accès lents, telles qu'un Raspberry Pi.

Les isolats *exe* sont, quant à eux, traités comme des boites noires, et n'intègrent donc pas d'éléments de la plate-forme.

La figure 5.1 présente la vue globale d'une application gérée par Cohorte Runtime.

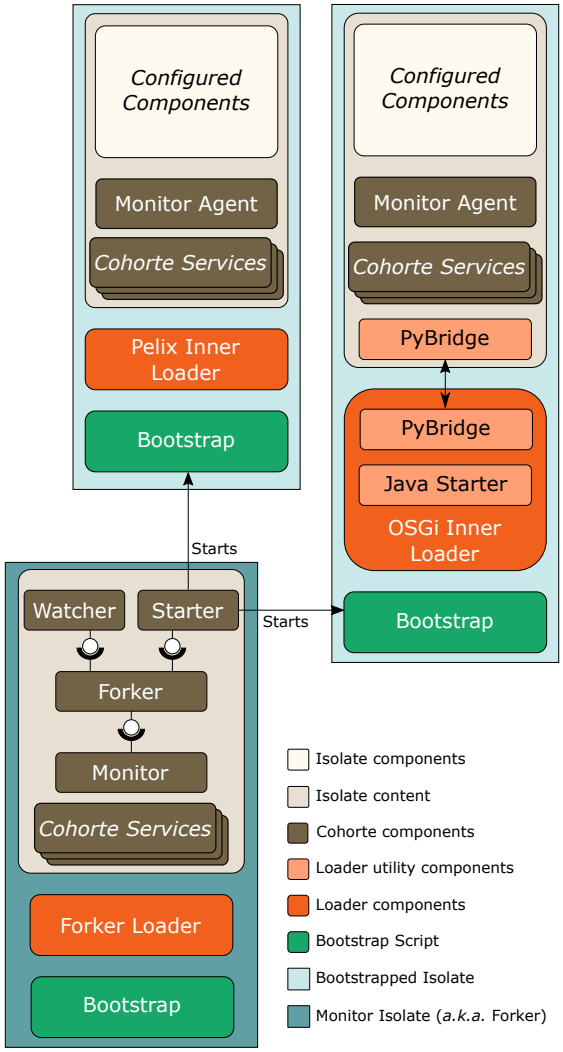

FIGURE 5.1 – Vue globale de Cohorte Runtime

5.5 Cycle de vie d'un isolat

L'état des composants est géré par le composant *status* du *Monitor* de l'isolat moniteur. Celui-ci associe à chaque isolat une machine à état fini, mise à jour à chaque étape du démarrage de l'isolat et suite aux événements provenant de l'isolat lui-même ou du *Forker*, *e.g.* lors de la notification du crash de l'isolat. Lorsqu'un isolat est déclaré perdu (suite à une défaillance, un arrêt imprévu, ...) ou bloqué (*freeze*), *i.e.* s'il ne répond plus aux requêtes, alors il est automatiquement redémarré par le moniteur, en utilisant la même configuration. Il est important de noter que la plate-forme Cohorte Runtime gère les isolats et instancie leurs composants, mais elle ne se charge pas du redémarrage de ces composants : c'est à la plate-forme Cohorte Composer, décrite dans le chapitre suivant, que revient cette tâche.

Dans cette section, nous présentons les différentes étapes du cycle de vie d'un isolat.

5.5.1 Démarrage d'un isolat par *bootstrap*

Le démarrage d'un isolat par *bootstrap* permet à Cohorte Runtime de configurer et de suivre finement son initialisation.

Fonctionnement général

Lorsque le *forker* doit démarrer un isolat par *bootstrap*, il rend sa configuration disponible à travers son serveur HTTP en la donnant au composant *broker*.

Le *bootstrap* est un script Python exécutable que le *forker* démarre. Ce dernier donne en paramètre du script l'identifiant unique et le nom de l'isolat ainsi que les URL vers sa configuration et vers le gestionnaire d'état des isolats. Le script utilise cette dernière pour indiquer chaque entrée dans une nouvelle étape du démarrage. Si cela est nécessaire, le *forker* indique un paramètre *looper*, décrit ci-après.

Le *bootstrap* commence par le démarrage d'une plate-forme Pelix et configure les propriétés initiales de celle-ci, en fonction des paramètres indiqués. Si le paramètre *looper* est présent, le *bootstrap* installe le bundle ayant le nom indiqué, le démarre puis attend l'apparition de son service `cohorte.boot.looper`. Ce dernier permet de démarrer une boucle de traitement d'événements dans le fil d'exécution principal du script. Si un *looper* est utilisé, la suite du chargement continue dans un nouveau fil d'exécution. Cette méthode a été mise en place principalement pour résoudre les problèmes liés aux composants disposant d'une interface graphique. En effet, la plupart des bibliothèques d'interfaces graphiques (Cocoa, Qt, ...) impose que leurs méthodes soient appelées depuis le fil d'exécution principal.

Le *bootstrap* installe ensuite un bundle de chargement, appelé *loader*. Cohorte Runtime supporte quatre loaders :

— `forker` : il est utilisé lorsque le script *bootstrap* est démarré sans indication d'URL de configuration. Il prépare l'isolat moniteur ;

— `broker` : il est utilisé quand une URL de configuration est indiquée. Il récupère cette configuration auprès du *broker* de l'isolat moniteur, puis charge le *loader* qui sera réellement mis en œuvre, c'est-à-dire l'un des deux suivants ;

— `pelix_inner` : le terme *inner* est employé pour montrer que le chargement de l'isolat continue dans la plate-forme démarrée par le *bootstrap*, dans laquelle le *loader* s'exécute ;

— `osgi_inner` : ce *loader* démarre une plate-forme OSGi Java dans l'interpréteur Python du *loader*. Il sera décrit plus en détail dans la section suivante.

Étant donné que les *loaders* sont fournis par des bundles, il est possible d'en définir d'autres, afin d'étendre la plate-forme. Ces *loaders* doivent avoir le même nom que le nouveau type (*kind*) qui est utilisé lors de la configuration des isolats.

Chaque *loader* lit les fichiers de configuration décrivant le contenu minimal d'un isolat, c'est-à-dire la liste des artefacts et des composants nécessaires au fonctionnement et à la surveillance de l'isolat ainsi qu'à la communication inter-isolats, *i.e.* les composants de Cohorte Herald. Le chargement est considéré terminé dès que l'agent du moniteur (*Monitor Agent*) a été instancié, validé, et qu'il peut communiquer avec l'isolat moniteur afin de lui indiquer son statut, *via* un service distant.

Cas des isolats Java

Un isolat Java est démarré à l'aide du *loader* `osgi_inner`. Celui-ci démarre une machine virtuelle Java (JVM) à l'intérieur de l'interpréteur Python, à l'aide de la bibliothèque JPype [1]. JPype est un module Python hybride, *i.e.* développé en C et en Python, qui permet de créer et d'interagir avec une JVM depuis Python. Il simplifie grandement ces actions en masquant les appels à l'API Java Native Interface (JNI) par une API de plus haut niveau.

Le *loader* démarre une plate-forme OSGi dans la machine virtuelle Java, dès que celle-ci est prête. Par défaut, il s'agit d'Apache Felix, mais il est possible de configurer d'autres implémentations, telles qu'Eclipse Equinox, Knopflerfish, etc.. Suivant les fichiers de configuration, le *loader* installe et démarre ensuite l'ensemble des bundles OSGi dans cette plate-forme. Il enregistre ensuite un service *bridge*, faisant la passerelle entre les environnements Python et Java. Ce service est implémenté en Python et ses méthodes respectent les signatures spécifiées par une interface Java : il est accessible par n'importe quel composant de la plate-forme Java. La prise en charge des appels à ce service, c'est-à-dire la conversion des arguments et de la valeur de retour, est assurée par JPype. Enfin, le *loader* installe et démarre le bundle Java qui doit instancier les composants Java. Celui-ci récupère la configuration de l'isolat auprès du *loader*, à l'aide du service *bridge*. Le

1. `https://github.com/tcalmant/jpype-py3`

chargement de l'isolat est considéré comme terminé quand l'agent Java du moniteur est démarré et peut communiquer avec l'isolat moniteur.

Comme dans le cas général, un isolat Java héberge le contenu minimal d'un isolat nécessaire à sa surveillance par l'isolat moniteur et à sa communication avec le reste de l'application. La partie Python de l'isolat reste accessible à travers le shell distant de Pelix. De cette manière, un opérateur peut contrôler l'état de l'isolat côté Python, voire ajouter des sondes, afin de surveiller la machine virtuelle Java.

5.5.2 Démarrage d'un isolat *exe*

Le démarrage d'un isolat *exe* est beaucoup plus simple car il n'est pas nécessaire de maîtriser son initialisation. Le *forker* démarre l'isolat comme un processus système à l'aide du module `subprocess`. Il peut ainsi configurer ses variables environnement et lui donner des arguments, tels qu'indiqués par l'opérateur de l'application dans la configuration de l'isolat.

La surveillance d'un isolat *exe* ne s'applique qu'au niveau du processus : l'isolat moniteur vérifie régulièrement si le processus s'est arrêté ou est notifié par le système de cet arrêt. Il n'est cependant pas possible de vérifier si ce processus est bloqué (*freeze*) ou s'il est dans un état invalide. Enfin, il n'est généralement pas prévu de communiquer avec ce type d'isolat de la même manière qu'avec les isolats démarrés par *bootstrap*, du fait de l'absence des composants de Herald.

Ce type d'isolat est généralement utilisé pour décrire des éléments tiers sur lesquels repose l'application, tels qu'un serveur LDAP, un service propriétaire aux sources fermées, etc.. Le fait qu'il soit géré par Cohorte Runtime permet de s'assurer que le processus soit redémarré suite à un arrêt manuel ou à un crash.

5.5.3 Disparition d'un isolat

Qu'il soit démarré par le *bootstrap* ou de type *exe*, un isolat est toujours surveillé à travers l'état de son processus. Le composant *IsolateWatcher* de l'isolat moniteur appelle régulièrement et de manière non bloquante la méthode POSIX `wait()`, ou son équivalent Windows `WaitForObject()`, pour vérifier si le processus de chaque isolat est toujours présent. La sélection de la méthode à appeler est effectuée au démarrage de l'isolat moniteur, en fonction du système d'exploitation hôte détecté. Si l'un de ces processus est absent, alors l'isolat moniteur considère que l'isolat surveillé s'est arrêté autrement que sur ordre de l'isolat moniteur, *e.g.* suite à un crash, à un arrêt programmatique (appel de la méthode `exit()`, ...) ou à un arrêt manuel (action de l'opérateur, de l'utilisateur, ...), et le considère alors disparu. Étant donné que l'*IsolateWatcher* sonde l'état des processus des isolats au lieu d'être notifié de leur disparition, il existe un délai entre l'arrêt d'un processus et sa détection.

Lorsque la plate-forme Cohorte Runtime est démarrée en mode *debogage*, une sonde supplémentaire est mise en place. Dans ce mode, l'isolat moniteur, toujours à travers son composant

IsolateWatcher, récupère et enregistre le contenu des sorties standard (*stdout*) et d'erreur (*stderr*) de chaque isolat surveillé. De cette manière, un opérateur peut analyser les journaux de ces isolats mais aussi les traces qui devraient normalement être affichées dans leur console, telles que des traces d'exceptions non gérées, etc.. Lorsqu'un processus se termine, ses sorties standard et d'erreur sont automatiquement fermées par le système d'exploitation hôte, entraînant un indication de fin de fichier. Cette dernière est alors détectée par l'isolat moniteur, qui va déclencher une vérification de l'état du processus à l'aide de la méthode `wait()`, afin de confirmer la disparition de l'isolat. Cette étape est nécessaire car certains isolats *exe* peuvent fermer leur sortie standard, notamment lorsqu'ils la redirigent vers un fichier sur disque. Elle permet également de réduire le délai entre la disparition d'un isolat et sa détection.

Lorsqu'un isolat est déclaré disparu, l'*IsolateWatcher* notifie ses abonnés de cet événement. Ce composant fonctionne suivant le modèle du *Whiteboard Pattern* : chaque composant souhaitant être notifié de ces disparitions doit fournir un service `watcher.listener`. À l'intérieur de Cohorte Runtime, seuls les composants *Forker* et *Monitor* écoutent ces événements. Lorsque le *Forker* est notifié de la disparition d'un isolat, il nettoie toutes les références correspondantes et s'assure de l'arrêt de tous les mécanismes de surveillance associés. Le *Monitor* vérifie quant à lui s'il s'agit d'un isolat qui a été configuré par l'opérateur (par un fichier de configuration) ou bien s'il a été démarré programmatiquement. Dans le premier cas, le *Monitor* se charge de demander au *Forker* de relancer l'isolat. Étant donné que l'isolat suit une configuration, il est redémarré avec le même contenu et les mêmes propriétés. Seul son identifiant unique est modifié. Dans le second cas, c'est à l'initiateur de l'isolat de s'abonner à cet événement et de demander au *Monitor* ou au *Forker* de le redémarrer. Cohorte Composer, présenté dans le chapitre suivant, se place dans le second cas.

5.5.4 Arrêt de la plate-forme

L'arrêt d'une plate-forme Cohorte Runtime s'effectue en trois étapes. Il est initié par un appel au composant *Monitor*, qui commence par positionner l'état de l'isolat moniteur dans le mode *arrêt de plate-forme*. Cet état empêche notamment le *Forker* de démarrer de nouveaux isolats.

Ensuite, le composant *Monitor* émet l'événement *arrêt de plate-forme* à tous les isolats démarrés par *bootstrap*. Ceux-ci peuvent alors initier l'arrêt de leur plate-forme d'exécution (OSGi, Pelix, ...). Le *Monitor* demande ensuite au *Forker* l'arrêt des isolats *exe*, qui correspond à l'envoi d'un signal `SIGKILL` (au sens POSIX). L'isolat moniteur est notifié de l'arrêt des isolats par l'*IsolateWatcher*. Si l'un d'entre eux met plus de temps à s'arrêter qu'un délai fixé par l'opérateur (par défaut, 5 secondes), alors le *Forker* le détruit à l'aide d'un signal `SIGKILL`.

Enfin, l'isolat moniteur s'arrête à partir du moment où tous les isolats ont été arrêtés ou détruits.

5.6 Conclusion

Dans ce chapitre, nous avons présenté la plate-forme Cohorte Runtime. Il s'agit de la partie technique de Cohorte assurant le démarrage et la surveillance d'isolats. Cette plate-forme se charge de leur redémarrage s'ils s'arrêtent inopinément (crash, arrêt manuel, ...). Cohorte Runtime peut démarrer des isolats à partir de fichiers de configuration ou suite à des ordres donnés programmatiquement par des outils externes tel que Cohorte Composer, présenté dans le chapitre suivant. Ces isolats peuvent être décrits de deux façons : soit (*i*) à partir de jeux d'artefacts et de composants, auquel cas il s'agit d'isolats initialisés par la plate-forme à l'aide d'un script *bootstrap*, soit (*ii*) à partir d'un ensemble de paramètres et de variables d'environnement afin d'exécuter un isolat *exe*, c'est-à-dire un processus *boîte noire* sur lequel la plate-forme n'a que peu de contrôle.

L'utilisation de primitives de bas niveau dépendantes du système d'exploitation hôte, telles que les méthodes `wait()` et `WaitForObject()`, a montré que le langage Python était efficace pour ce genre d'opération. En effet, ce dernier permet un accès simple à ces méthodes, sans avoir à fournir une version spécifique au système d'exploitation cible : la détection de la méthode à utiliser est effectuée à l'exécution, durant le démarrage de la plate-forme.

La table 5.1 présente le nombre de lignes de code (LOC) contenues dans Cohorte Runtime, en Python et en Java. La première ligne indique le nombre de lignes de code que compte l'intégralité de Cohorte Runtime, les lignes suivantes détaillent certains groupes de ses composants. Seules les lignes contenant des instructions sont comptées ; les commentaires, la documentation et les lignes vides sont omis. Le nombre de lignes de code a été mesuré à l'aide de l'utilitaire *cloc.pl*[2].

Faisant partie de la plate-forme Cohorte, le projet Cohorte Runtime est un projet libre diffusé sous les termes de la licence Apache Software License 2.0[3] (ASL 2.0). Contrairement à iPOPO et Cohorte Herald, il n'est pas disponible comme une bibliothèque indépendante mais il fait partie du livrable Cohorte[4]. Son code source est disponible sur la forge en ligne GitHub[5].

2. `http://cloc.sourceforge.net/`
3. `http://www.apache.org/licenses/LICENSE-2.0`
4. `http://cohorte.github.io/downloads/`
5. `https://github.com/isandlaTech/cohorte-runtime`

Cohorte Runtime	LOC Python	LOC Java	Python+Java
Total Cohorte Runtime	6 441	4 899	11 340
Bootstrap (base)	293	0	293
Configuration	468	0	468
Loopers	270	0	270
cocoa	*37*	*0*	*37*
qt	*62*	*0*	*62*
Loaders	623	311	934
forker	*113*	*0*	*113*
broker	*168*	*0*	*168*
pelix_inner	*33*	*0*	*33*
osgi_inner	*341*	*311*	*652*
Monitor	363	0	363
Monitor Agent	59	548	607
Forker	802	0	802
starters	*254*	*0*	*254*
watchers	*104*	*0*	*104*
Repositories	1332	0	1332
Python	*451*	*0*	*451*
Java	*579*	*0*	*579*

TABLE 5.1 – Lignes de code de Cohorte Runtime

CHAPITRE 6

CONTRIBUTION 4 : COHORTE COMPOSER

“ Every program is a part of some other program and rarely fits. ”

Alan J. PERLIS, *Epigrams on Programming*, 1982

Sommaire

6.1 Introduction 93
6.2 Architecture 94
6.3 Modèle de composition 96
6.3.1 Composition 96
6.3.2 Composant 97
6.3.3 Comparatif 98
6.4 Implémentation 99
6.4.1 Surveillance et notation des composants 99
6.4.2 Distribution des composants par vote 100
6.4.3 Instanciation des composants 103
6.5 Gestion autonomique de la composition 105
6.5.1 Définition 105
6.5.2 Comparaison avec Cohorte Composer 107
6.6 Conclusion 108

6.1 Introduction

Ce chapitre présente Cohorte Composer, un ensemble de composants se chargeant de la gestion des compositions d'une application répartie. Ces composants sont intégrés aux isolats moniteurs de différentes instances de la plate-forme Cohorte Runtime. Ils interagissent avec cette plate-forme afin d'instancier les composants d'une application dans plusieurs isolats, sur plusieurs machines. Pour mémoire, la plate-forme Cohorte Runtime, présentée dans le chapitre précédent, a pour objectif de simplifier la répartition d'une application en un ensemble d'isolats, *i.e.* de processus, au sein d'une même machine.

Cohorte Composer permet de décrire une application sous la forme d'un ensemble de compositions. Chaque composition est un ensemble de composants associés à leur configuration. Ces

composants sont regroupés en fonction de leur langage d'implémentation, de leur configuration et de leur fiabilité, chaque groupe étant ensuite instancié dans un isolat. Par rapport à la plate-forme Cohorte Runtime, Cohorte Composer prend en charge l'instanciation et la surveillance des composants de l'application, et non des isolats les hébergeant.

Nous présentons dans ce chapitre l'architecture globale de Cohorte Composer, son modèle de composition et enfin les différents éléments propres à notre implémentation.

6.2 Architecture

Cohorte Composer se charge de lire les descriptions de compositions, de répartir ses composants dans différents isolats, sur différentes machines (appelées nœuds), de les instancier et enfin de les surveiller. Lorsqu'un composant disparait d'un isolat, suite à une suppression manuelle de l'instance ou à l'arrêt de son artefact, Cohorte Composer l'instancie à nouveau, si nécessaire après avoir attendu la réapparition de sa fabrique. Si un isolat hébergeant des composants disparait, alors une nouvelle distribution de ces derniers sera calculée afin de les placer dans de nouveaux isolats ou dans ceux déjà présents. Ce nouveau calcul ne prend en compte que les composants perdus : les autres composants ne sont pas concernés. Pour effectuer ces différentes tâches, nous avons structuré Cohorte Composer en trois niveaux : le *Top Composer*, le *Node Composer* et l'*Isolate Composer*. La figure 6.1 montre la répartition de ces niveaux dans une application.

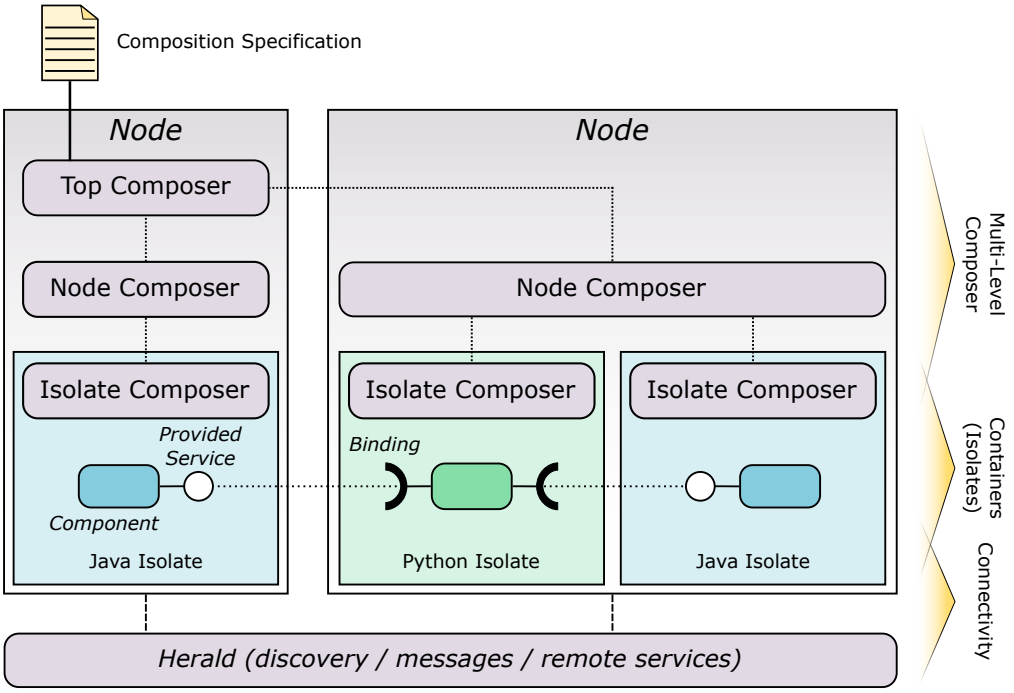

FIGURE 6.1 – Vue globale de Cohorte Composer

Le *Top Composer* est le point d'entrée unique pour démarrer une application. Il en a une connaissance globale et prend en charge toutes les compositions. Pour chacune d'entre elles, il calcule le placement de ses composants, selon leur configuration, sur les différents nœuds (*nodes*) de l'application. Le *Top Composer* peut alors demander la création des nœuds, s'il a accès à un tel service, puis attend l'apparition des *Node Composers*. Ces derniers sont détectés à travers le service qu'ils fournissent. Étant donné que Cohorte Composer est un ensemble de composants intégrés à Cohorte Runtime, leurs services sont exportés, découverts et importés par Cohorte Herald.

Un *Node Composer* doit être présent sur chaque nœud, *i.e.* chaque machine, qui héberge une partie de l'application. Dès qu'un *Node Composer* est détecté par le *Top Composer*, ce dernier lui transmet la liste des composants à instancier au sein du nœud dont il est responsable, ainsi qu'une copie de la répartition calculée pour l'ensemble de l'application. Cette copie permet de rendre le *Top Composer* optionnel : il peut être arrêté une fois le calcul effectué et les ordres d'instanciation transmis aux nœuds. Le *Top Composer* pourra récupérer ultérieurement cette configuration pour calculer un nouveau placement, par exemple si la composition de l'application doit être mise à jour. Le *Node Composer* a pour tâche principale le calcul du regroupement des composants en fonction de leur langage d'implémentation, des indications données par l'opérateur dans la composition et de critères mesurés par des composants d'extension. Ces derniers peuvent par exemple associer à chaque composant une note de fiabilité. Ce mécanisme dépend de l'implémentation de Cohorte Composer et est décrit dans la section 6.4. Contrairement au *Top Composer*, dont la distribution des composants est calculée une seule fois pour chaque version d'une composition, le *Node Composer* recalcule la sienne régulièrement et en réponse à des événements, tels que l'arrêt d'un isolat ou la détection du comportement erratique d'un composant. Chaque groupe de composants sera ensuite assigné à un isolat : soit celui indiqué dans la composition, soit un nouvel isolat, soit un isolat existant. Le *Node Composer* utilise le service fournit par le *Monitor* de l'isolat moniteur de son nœud pour démarrer les isolats supplémentaires. Suivant le même mécanisme que le *Top Composer*, le *Node Composer* attend ensuite l'apparition des différents services *Isolate Composer* pour leur donner leurs ordres d'instanciation des composants qui leur ont été assignés.

Enfin, l'*Isolate Composer* est présent dans tous les isolats démarrés sur demande d'un *Node Composer*. Il reçoit du *Node Composer* la liste des composants à instancier, leur configuration ainsi que la liste des artefacts fournissant leurs fabriques. L'*Isolate Composer* installe puis démarre ces artefacts et regroupe les composants en fonction de leur modèle (iPOJO, iPOPO, ...). Il transmet ensuite ces groupes aux agents correspondants. Ces derniers ont pour tâche de transformer la description des composants en une description conforme au modèle du composant. Pour cela, ils préparent les paramètres de démarrage des composants et convertissent les descriptions de leurs liaisons (filtres et fils). Ensuite, ils instancient ces composants dès que leur fabrique est disponible. Les agents doivent également réinstancier un composant s'il disparait, par exemple si un utilisateur détruit manuellement une instance de composant ou si l'artefact fournissant sa fabrique est mis à

jour. Cependant, dans le cas où un artefact est désinstallé manuellement, l'*Isolate Composer* se met en attente de la réapparition d'une fabrique du même nom : il ne se chargera pas de réinstaller l'artefact concerné, considérant qu'il peut s'agir d'une opération de maintenance.

Dans le cas des composants orientés service, les liaisons entre ceux-ci sont assurées par les services distants gérés par Cohorte Herald. En effet, sauf mention contraire, l'*Isolate Composer* fait en sorte que tous les services de tous les composants soient exportés par ce biais. Pour les autres types de modèles, il est recommandé de regrouper leurs composants dans le même isolat, car la plate-forme ne prend pas en charge leurs liaisons inter-isolats.

Le *Top Composer* peut être exécuté sur le même nœud qu'un *Node Composer*. Ce nœud sera utilisé pour héberger les composants n'ayant aucune indication d'emplacement d'exécution. Cependant, le *Top Composer* peut également être considéré comme un élément amovible : il ne doit être présent qu'au moment du chargement de la description de la composition et de l'apparition d'un nouveau nœud. En effet, les *Node Composers* ne remontent aucune information vers le *Top Composer*. Enfin, une fois ses composants reçus, chaque *Node Composer* devient alors autonome et apte à gérer sa composition en autarcie.

6.3 Modèle de composition

Les descriptions de compositions et de composants utilisées par Cohorte, ont été définies selon les informations pouvant être données aux plates-formes d'exécution iPOJO et iPOPO. Bien que Cohorte Composer ait été initialement développé pour ces deux modèles à composants orientés service (SOCM), son architecture permet de l'étendre afin de gérer d'autres SOCM, voire d'autres types de modèles à composants, en ajoutant de nouveaux agents à l'*Isolate Composer*. De plus, le modèle de composant utilisé, décrit ci-après, permet pour cela la configuration de liaisons à partir de l'indication d'un nom de composant cible.

6.3.1 Composition

La description d'une composition d'une application Cohorte est une liste hiérarchisée de composants. Une composition a un nom, permettant de l'identifier à l'exécution, et un composite racine. Un composite est un ensemble, nommé, de composants élémentaires et de composites enfants. Les liaisons inter-composants sont optionnelles, configurables dans la description du composant consommateur. À titre de comparaison, le modèle SCA les définit au niveau du composite.

À l'exécution ce modèle est mis à plat, *i.e.* la hiérarchie de composites est transformée en un seul ensemble de composants. Le nom complet d'un composant est alors le résultat de la concaténation du nom de sa composition, des noms de ses composites parents et de son nom. En comparaison, le composite racine du modèle Cohorte correspond au domaine du modèle SCA [SCA11], et les sous-composites Cohorte sont proches des composites SCA.

6.3.2 Composant

Chaque composant d'une composition Cohorte Composer est défini par au moins un nom de type et un nom d'instance. Le nom de type permet de déterminer quelle classe ou quelle fabrique de composant doit être utilisée pour instancier le composant. Dans les modèles iPOJO et iPOPO, il correspond au nom de la fabrique (*factory*) du composant. Le nom du composant est la partie terminale du nom de son instance et il doit être unique dans un composite : le nom final du composant est la concaténation du nom complet du composite contenant le composant avec le nom de ce dernier. De cette manière, une instance de composant peut être identifiée facilement dans une application. Ce mécanisme permet d'avoir des composites et des composants portant le même nom simple dans une même application, s'ils appartiennent à différentes compositions. Par exemple, si deux compositions `C1` et `C2` définissent chacune un composant ayant pour nom simple `Exemple`, les noms complets des composants deviendront `C1.Exemple` et `C2.Exemple`.

Un composant peut également avoir des propriétés initiales. Celles-ci sont définies à titre de configuration du composant, bien que son implémentation puisse l'ignorer. Elles peuvent aussi servir à configurer le comportement de l'agent en charge d'instancier le composant, par exemple pour indiquer que seule une partie des services fournis par le composant peut être exportée vers des machines distantes.

Les liaisons du composant, aussi appelées exigences, sont généralement laissées à la discrétion de la plate-forme d'exécution de son modèle, mais peuvent également être configurées par l'opérateur de l'application. Pour cela, le modèle du Cohorte Composer dispose de deux éléments de description : les filtres (*filters*) et les fils (*wires*). Les premiers permettent d'associer un filtre LDAP à une exigence du composant. Cette exigence est décrite par un identifiant ayant une signification pour le modèle à composants sous-jacent. Le filtre LDAP sera utilisé pour sélectionner les services correspondant à l'exigence, en fonction des valeurs de leurs propriétés. Les fils associent, quant à eux, un nom de composant à une exigence. Ils permettent de forcer la liaison entre deux composants et peuvent également être utilisés pour configurer les liaisons entre composants d'un modèle non-orienté service. Dans les deux cas, ces informations sont transmises, sans modification, jusqu'à l'agent en charge d'instancier le composant. Ce dernier convertit alors ces informations afin qu'elles soient conformes au modèle à composants concerné.

Il est possible de fixer la position du composant dans l'application en spécifiant le nom de son nœud d'exécution ou celui de son isolat hôte. Dans le premier cas, le composant sera assigné au nœud indiqué et sera placé dans un isolat selon la décision de son Node Composer. Dans le second cas, le composant sera nécessairement placé dans l'isolat portant le nom indiqué. Si une configuration existe pour cet isolat, indiquant un nœud d'exécution, alors le composant sera instancié sur ce nœud, dans cet isolat. Si, au contraire, elle n'existe pas ou n'indique pas de nœud, alors le composant est assigné à un nœud par le Top Composer et sera exécuté dans l'isolat indiqué.

Enfin, l'opérateur de l'application peut également indiquer des informations sur le fournisseur

du type de composant. Le nom de fichier de l'artefact fournissant le composant peut être forcé afin, par exemple, d'en utiliser une version particulière, notamment si plusieurs versions sont disponibles. L'indication du nom de fichier permet aussi d'assister Cohorte Composer en précisant une information non automatiquement détectable, *e.g.* si aucune métadonnée ne permet d'associer à un artefact les types de composants qu'il fournit. Le langage d'implémentation du type de composant peut lui aussi être indiqué afin de simplifier ou limiter la recherche de l'artefact associé. Cette indication permet de sélectionner la version souhaitée d'un type de composant disponible dans différents langages, par exemple pour tester une nouvelle implémentation dans un langage plus approprié.

6.3.3 Comparatif

Description	**Cohorte**	**Fraclet**	**SCA**
Type de composant	Nom de fabrique iPOJO/iPOPO	*Non applicable*	Implémentation ou type de composant
Instance de composant	Instance de la fabrique	Composant	Composant
Identifiant du composant	Nom d'instance	Nom de composant	Nom d'instance
Description des exigences	Implicite, décrite selon le SOCM sous-jacent et guidée par les filtres et les fils	Liste d'interfaces de composants ou de services de la plate-forme	Liste de références, guidées au niveau du conteneur du composant
Description des services fournis	Implicite, décrite selon le SOCM sous-jacent	Liste d'interfaces	Liste de services
Propriétés du composant	Dictionnaire de propriétés initiales	Liste d'attributs	Liste de propriétés

TABLE 6.1 – Comparaison des descriptions de composants Cohorte, Fraclet et SCA

Les informations utilisées par Cohorte Composer sont génériques et peuvent être comparées aux informations utilisées par les modèles Fraclet [RM09] et SCA [SCA11]. Le tableau 6.1 présente les différences entre ces deux modèles et celui de Cohorte Composer. L'utilisation de liaisons implicites, point différenciant le plus fortement le modèle de Cohorte des autres modèles présentés, est une capacité à double tranchant. En effet, celle-ci permet de simplifier énormément la description de la composition, en indiquant seulement la liste des composants à instancier, sans décrire précisément leurs interactions. Cependant, cette capacité ne permet pas d'avoir une vision globale de l'application avant qu'elle ne soit exécutée, ni de vérifier si une composition est correcte, c'est-à-dire si toutes les exigences de chaque composant seront satisfaites à l'exécution. Néanmoins, la validation d'une description de composition ne garantie pas la validité de ses composants à l'exécution : certains de ces derniers peuvent en effet dépendre de services théoriquement fournis par la plate-forme ou par des artefacts mais absents à l'exécution, *e.g.* du fait de conditions inhérentes à leur implémentation.

6.4 Implémentation

Les sections précédentes ont présenté le modèle de Cohorte Composer et son fonctionnement général. Cette section présente l'implémentation que nous proposons. Elle s'appuie sur des choix spécifiques concernant certaines phases du fonctionnement du Composer. Ces dernières pourraient être développées différemment, soit dans une autre implémentation soit en tant qu'alternative ou complément de l'implémentation de référence.

6.4.1 Surveillance et notation des composants

Cohorte Composer a pour objectif de maintenir en activité la composition d'une application : l'ensemble de tous ses composants doit être instancié et rester valide le plus longtemps possible. Cohorte Composer a pour but de rechercher une qualité de service (QoS) optimale de l'application.

Pour cela, Cohorte Composer nécessite d'être notifié de deux types événements : (*i*) la disparition d'un composant et (*ii*) la suspicion d'erreur dans un composant. La disparition d'un composant peut avoir deux origines : soit il a été détruit par un tiers (utilisateur, opérateur, composant de l'application, ...), soit l'isolat l'hébergeant s'est arrêté de fonctionner. Dans le premier cas, c'est à l'*Isolate Composer* de réagir : le composant sera à nouveau instancié dès que sa fabrique sera opérationnelle. Dans le second cas, le *Forker* de Cohorte Runtime détecte l'arrêt de l'isolat et en notifie le *Node Composer*. Le comportement erroné d'un composant est, quant à lui, indiqué auprès du *Node Composer* par délation de la part de l'un de ses consommateurs. Cette méthode permet notamment d'être notifié du gel (*freeze*) d'un composant, *e.g.* s'il est en situation d'interblocage (*deadlock*), et de déclencher son redémarrage voire sa migration dans un isolat spécifique. Le consommateur peut émettre cet événement si, par exemple, le service qu'il consomme ne respecte pas la qualité de service attendue ou négociée, *e.g.* s'il dépasse le temps d'exécution d'une opération [Tou10, Amé13].

Lorsqu'un événement concernant un composant atteint le *Node Composer*, celui-ci déclenche un nouveau calcul de distribution pour tous les composants de l'isolat impliqué. Comme il a été indiqué plus haut, ce calcul s'appuie sur un certain nombre de critères, dépendant de l'implémentation. Dans notre implémentation, nous utilisons trois critères :

— le premier critère se base sur la description de la composition, afin de forcer l'hébergement d'un composant dans l'isolat indiqué par l'opérateur ;

— le deuxième critère associe à chaque composant une note de fiabilité ;

— le troisième critère conserve l'historique des crashs, *i.e.* la liste des composants qui étaient hébergés dans un isolat qui a été déclaré défaillant.

La note de fiabilité a été le premier critère implémenté dans Cohorte Composer. Il s'agit d'une valeur entière qui est fortement dégradée lorsque le composant disparait ou s'il est déclaré erroné. En contrepartie, cette valeur est légèrement augmentée après une certaine période de temps

sans événement négatif afin d'éviter de considérer non fiable, par erreur, un composant subissant l'instabilité de ses voisins. L'interprétation de cette note est guidée par des paliers :

— si elle dépasse une valeur de confiance, alors le composant est considéré fiable et peut être regroupé avec les autres composants qualifiés de cette manière ;

— à l'inverse, si elle passe sous le seuil de méfiance, le composant sera placé dans un des isolats contenant des composants équivalents ;

— enfin, si le seuil d'instabilité est franchi, le composant sera hébergé dans un isolat qui lui sera propre (isolation individuelle), afin que les pannes qu'il provoque n'influent que légèrement sur le reste de l'application.

Une version plus avancée de ce critère a été développée afin d'évaluer la fiabilité d'un composant en fonction de ses voisins, *i.e.* des composants présents dans le même isolat lors d'un événement négatif. Elle a cependant été abandonnée lors de sa phase de prototypage car elle convergeait rapidement vers l'isolation individuelle de chaque composant. Cette méthode de notation a cependant inspiré le critère basé sur l'historique des crashs.

L'historique des crashs permet quant à lui d'éviter de retrouver, dans une distribution, des combinaisons de composants ayant déjà subi un crash. Lorsqu'un isolat s'arrête, les composants qu'il héberge sont suspectés d'être incompatibles et donc d'être la cause de cet arrêt. À chaque nouveau crash, ce critère cherche à réduire la taille des listes conservées : si le groupe de composants concerné par ledit crash est un sous-ensemble d'un groupe déjà connu, alors seul ce sous-ensemble est conservé. L'objectif de cette technique est d'obtenir une information fine sur les incompatibilités entre composants à partir des arrêts qu'ils ont causés.

En liant l'historique des crashs et la note de fiabilité, nous avons observé que le système converge vers des groupes stables de composants, c'est-à-dire suffisant fiables pour ne pas être à nouveau scindés.

6.4.2 Distribution des composants par vote

Dans le cadre de Cohorte Composer, le calcul d'une distribution s'effectue à travers une suite de votes. Pour chaque composant à distribuer, le *Node Composer* lance un nouveau vote ayant pour sujet ledit composant et ayant pour candidats les isolats présents sur le même nœud. Un candidat supplémentaire est également proposé, représentant un isolat vierge, *i.e.* l'isolat à utiliser si aucun de ceux présents n'est acceptable. Tous les critères de distribution, aussi bien ceux intégrés à Cohorte Composer que ceux fournis par des extensions, sont les électeurs :

— le critère de compatibilité interdit le regroupement de composants développés dans des langages d'implémentation différents, *e.g.* l'un en Java et l'autre en Python ;

— le critère de configuration force, par "coup d'état", l'hébergement d'un composant dans l'isolat indiqué par l'opérateur. Si aucun isolat n'est indiqué, cet électeur vote blanc ;

— l'historique de crash et la note de fiabilité votent quant à eux pour les isolats correspondant au mieux à leurs exigences.

Lors d'un vote, les électeurs connaissent la liste des composants qui ont été assignés à chaque isolat durant cette distribution, ainsi que l'isolat qui hébergeait le composant sujet lors de la distribution précédente. Ces informations permettent aux critères de préférer ce dernier aux autres candidats afin d'éviter une migration superflue du composant vers un autre isolat.

Le moteur de vote que nous avons développé appelle chaque service électeur en lui indiquant la liste des candidats pour lui faire remplir un bulletin de vote, puis consomme un service de dépouillement, correspondant à un système de vote, afin d'obtenir un résultat. Ce résultat peut être un candidat ou un ensemble de candidats. Ce moteur prend en compte les "coups d'état", c'est-à-dire le fait qu'un électeur force le résultat d'une élection. Chaque élection est associée à un sujet, afin que les électeurs sachent selon quels critères ils vont voter. Un électeur peut indiquer sur son bulletin de vote une ou plusieurs approbations et désapprobations, avec un ordre de préférence dans ses choix.

Dans notre implémentation, le calcul de la distribution des composants est effectué selon le système de vote par *approbation* [1]. Dans ce système, chaque électeur vote pour *tous* les candidats qui correspondent à ses exigences, sans ordre de préférence. Selon les conditions de vote, *i.e.* selon la configuration du service de dépouillement, un électeur peut également voter contre un candidat. Au delà du seuil de désapprobation, un candidat peut être exclu. Le résultat du vote est le candidat ayant obtenu le plus de voix positives et le moins de voix négatives. En cas d'égalité, c'est le premier élément de la liste des candidats qui est déclaré vainqueur. Cette liste est préparée par le consommateur du système de vote, *i.e.* le *Node Composer*, et ce de façon à ce que son contenu soit toujours dans le même ordre. De cette manière, si le même cas d'égalité est rencontré plusieurs fois, le système de vote renverra toujours le même résultat, l'ordre de la liste des candidats restant stable.

Étant donné que les services de dépouillement sont configurables dans cette implémentation, il est possible de transformer le vote par *approbation* en un vote *pondéré* [2]. Dans ce cas, chaque électeur peut associer un poids à chaque candidat, ce qui équivaut à voter plusieurs fois pour lui. Pour des raisons d'équité, tous les électeurs disposent du même nombre de poids à placer sur leur bulletin, mais ils n'ont pas l'obligation d'utiliser l'intégralité de ces poids.

Outre le vote par *approbation* et le vote *pondéré*, nous avons également implémenté le vote par *majorité absolue* [3] et le vote *alternatif* [4]. Le vote par *majorité absolue* suit les règles de vote utilisées lors des élections présidentielles françaises. Chaque électeur vote pour un seul candidat, le

1. https://en.wikipedia.org/wiki/Approval_voting
2. https://fr.wikipedia.org/wiki/Vote_pond%C3%A9r%C3%A9
3. https://fr.wikipedia.org/wiki/Scrutin_uninominal_majoritaire_%C3%A0_deux_tours
4. https://en.wikipedia.org/wiki/Instant-runoff_voting

vainqueur étant celui ayant la majorité absolue des voix (50% des voix, plus une). En cas d'échec, un second vote est lancé en ne gardant que les candidats ayant obtenu le plus de voix. S'il débouche sur une égalité de voix, le premier élément de la liste des candidats est déclaré vainqueur (pour information, selon l'article L262 du Code électoral, le vainqueur désigné lors d'une égalité dans une élection présidentielle française est le candidat le plus âgé).

Le vote *alternatif* est un mode de scrutin prenant en compte l'ordre de préférence des électeurs. Chaque électeur peut voter pour tout ou partie des candidats, en indiquant sur son bulletin son ordre de préférence. Le premier résultat intermédiaire du vote est construit en comptant les voix correspondant au premier choix, *i.e.* de première préférence, de chaque électeur. Si un candidat obtient la majorité absolue, il sort vainqueur du vote, sinon un nouveau vote intermédiaire a lieu. Le candidat qui a obtenu le moins de voix est alors éliminé et ses voix ne sont plus prises en compte. Le rang de préférence de chaque bulletin est mis à jour afin de supprimer toute référence à ce candidat, *i.e.* ses voix sont transférées au candidat suivant dans chaque liste de préférence. Ce processus continue tant qu'aucun candidat n'obtient la majorité absolue. Dans notre implémentation de ce mode de scrutin, si un résultat intermédiaire débouche sur une égalité des voix, c'est le dernier élément de la liste des candidats qui est disqualifié.

L'utilisation d'un système de vote pour calculer la distribution des composants entre les isolats a été adoptée car elle assure l'obtention d'un résultat et ce dans un délai dépendant uniquement du nombre d'isolats sur le nœud et du nombre de composants à instancier. Cependant, d'autres pistes ont été envisagées mais n'ont pas dépassé le stade expérimental.

Des solveurs de problèmes de satisfaction de contraintes (*Constraint Satisfaction Problem*, noté CSP) avaient été mis en place avant l'adoption du système de vote. Cette implémentation du calcul de distribution était basée sur la bibliothèque Google's Operations Research tools [5]. Les critères de distribution remplissent alors une matrice de distances entre chaque composants afin de représenter, sous forme d'un graphe, les liaisons d'incompatibilités entre composants. Le problème à résoudre était inspiré de celui de la coloration de graphe : les composants liés entre eux et dont le poids de l'arête dépassait un seuil donné devaient avoir une *couleur* différente. La fonction d'optimisation consistait à utiliser un minimum de *couleurs*. Dans notre cas, une couleur représentait un isolat. Cette implémentation n'a pas été adoptée du fait du caractère non-déterministe des solveurs CSP. En effet, il n'est pas possible de prévoir si une solution à un problème CSP sera trouvée ni sous quel délai. Cependant, dans notre cas, le CSP peut toujours renvoyer la solution la plus simple et la moins efficace, *i.e.* celle où chaque composant est isolé individuellement. De plus, le non-déterminisme du solveur signifie que son résultat peut être et est généralement différent à chaque exécution, même si ses paramètres restent inchangés. Dans le cadre de Cohorte Composer, cela se traduit par la migration des composants d'un isolat à l'autre, et donc par de fortes perturbations des performances, après chaque calcul de distribution.

5. `https://developers.google.com/optimization/`

Pour cette raison, nous avons préféré l'utilisation d'un système de vote déterministe, c'est-à-dire que son résultat reste identique lorsqu'il est exécuté plusieurs fois avec les mêmes paramètres, dans le même contexte. L'opérateur peut sélectionner le système de vote qui lui semble le plus efficace dans la configuration des *Top* et *Node Composers*. Par défaut, c'est le vote par *approbation* qui sera utilisé du fait de sa prédictibilité. En effet, son résultat correspond toujours à l'isolat répondant aux attentes d'un maximum de critères et ne change que suite à des variations importantes des données des différents critères. À l'inverse, le vote par majorité absolue élimine les seconds choix de chaque électeur et ses résultats peuvent donc basculer du tout au tout assez rapidement. Ce mode de scrutin n'est pas recommandé à l'usage car il peut apporter un effet de cycle, c'est-à-dire faire migrer un composant d'un isolat à l'autre au cours des différentes distributions. Enfin, le vote *alternatif* offre des résultats beaucoup plus stables que ceux du vote par majorité absolue, mais sa mise en œuvre rend plus complexe le développement des critères électeurs car ceux-ci doivent être capables de donner un ordre de préférence sur leur bulletin.

6.4.3 Instanciation des composants

L'*Isolate Composer* est un composant générique qui supporte plusieurs modèles à composants. Il délègue les tâches concernant les composants à des agents. Chaque agent est un service associé à un et un seul modèle à composants et ne traite que les actions à effectuer sur des composants de ce modèle. Ainsi, lorsque le *Node Composer* demande l'instanciation d'un groupe de composants, l'*Isolate Composer* installe les artefacts fournissant leurs fabriques, répartit ce groupe en fonction du modèle de chaque composant et enfin transmet chaque groupe à l'agent correspondant. Cet agent compare le groupe de composants avec la liste de ceux qui ont déjà été instanciés, afin d'éviter la duplication d'instances déjà existantes. Pour chaque composant, il vérifie ensuite si sa fabrique est accessible, auquel cas il convertit sa description du modèle Cohorte Composer vers un format intelligible par le modèle à composants à utiliser puis instancie le composant. Si la fabrique n'est pas accessible, le composant est mis de côté et son instanciation sera à nouveau tentée au moment où l'agent sera notifié de l'apparition de la fabrique du composant. En cas d'arrêt imprévu du composant, c'est l'agent en charge de son modèle qui doit l'instancier à nouveau, si nécessaire en attendant la réapparition de sa fabrique. L'autre ordre que peut donner le *Node Composer* est l'arrêt d'un composant : l'agent est appelé avec le nom d'un composant et déclenche l'arrêt ce dernier suivant la procédure du modèle à composants sous-jacent.

L'implémentation actuelle de Cohorte Composer comporte deux agents, afin de prendre en charge les modèles à composants iPOPO et iPOJO. Bien que ces modèles soient conceptuellement proches, leur API d'instanciation des composants est différente.

iPOPO fournit un service `core` par lequel n'importe quel composant peut demander des informations sur le contenu de la plate-forme (fabriques disponibles, instances existantes, ...) ainsi que l'instanciation d'un composant. L'agent indique alors à iPOPO le nom de la fabrique à utiliser,

le nom de l'instance et ses propriétés initiales. Si la fabrique n'existe pas, une exception est levée et le composant n'est pas instancié. L'agent s'inscrit dès sa validation auprès du service `core` pour être notifié des événements iPOPO concernant les fabriques et les instances de composants (apparition, disparition, changement d'état, ...). De cette manière, l'agent est capable de tenter une nouvelle instanciation pour chaque composant dont la fabrique est (ré)apparue, mais aussi pour chaque composant ayant été détruit autrement que sur ordre du *Node Composer*, *e.g.* suite à la mise à jour de son bundle.

Pour sa part, le modèle à composants iPOJO fournit un service `Factory` pour chaque fabrique de composant qu'il peut utiliser. L'agent s'inscrit auprès du registre de services de la plateforme OSGi afin d'être notifié de l'apparition et de la disparition de ces services fabriques. Il instancie ensuite chaque composant en consommant le service `Factory` correspondant et en lui indiquant des propriétés initiales. Le nom de l'instance de composant fait partie de ces propriétés. Enfin, l'agent s'inscrit aux événements de chaque instance créée afin d'être notifié de ses changements d'état et de sa disparition.

Étant donné la proximité entre le modèle de Cohorte Composer et les modèles d'iPOPO et d'iPOJO, les agents n'ont pas de conversion complexe à effectuer pour transformer les représentations des composants d'un modèle à l'autre. Les fils (*wires*), forçant à effectuer les liaisons entre composants selon leur nom, sont convertis en un filtre LDAP testant la valeur de la propriété `instance.name` [6], celle-ci étant configurée par iPOJO et iPOPO. Quand c'est nécessaire, ils sont fusionnés avec les filtres LDAP configurés par l'opérateur (*filters*) à l'aide d'un opérateur `ET` (&). Ces filtres LDAP sont ensuite associés aux différentes dépendances à travers les propriétés `requires.filters` et `temporal.filters` des composants, utilisées par les *handlers* iPOPO et iPOJO pour surcharger leur configuration. Enfin, chaque agent ajoute la propriété `service.exported.interfaces` à tous les composants instanciés afin de s'assurer que leurs services seront exportés et donc accessibles par les composants hébergés dans d'autres isolats.

Cette architecture permet d'étendre l'*Isolate Composer* avec de nouveaux agents afin de gérer d'autres modèles à composants. L'implémentation d'un agent pour le modèle BluePrint [OSG13], spécifié par OSGi, est envisagée. Il en va de même pour la plupart des autres modèles spécifiés ou s'appuyant sur OSGi, tels que Declarative Services [OSG05a]. Enfin, en ce qui concerne le langage Python, nous avons pour projet de développer un agent pour le modèle Zope Component Architecture [7] (ZCA).

6. Par exemple : `(instance.name=nom-du-composant-cible)`
7. `http://docs.zope.org/zope.component/index.html`

6.5 Gestion autonomique de la composition

6.5.1 Définition

Le terme *autonomique*, tel que présenté par [KC03], fait référence au système nerveux autonome (ou système nerveux viscéral), un système nerveux périphérique des vertébrés. Il s'agit de la partie du système nerveux responsable des fonctions qui ne sont pas soumises à un contrôle volontaire. C'est le cas, par exemple, des fonctions cardiovasculaires, respiratoires et digestives. Il est notamment en charge de la gestion de l'énergie en période de stress (augmentation des rythmes cardiaque et respiratoire, ...).

Le concept d'*informatique autonomique* a été introduit par IBM en 2001 [Hor01], bien que des logiciels s'appuyant sur ses notions existaient déjà auparavant. Son objectif est de permettre à l'utilisateur d'une application de se focaliser sur sa tâche et non sur les techniques à employer pour la réaliser. Les applications autonomiques doivent ainsi être capables de réagir d'elles-mêmes lorsque des événements se produisent, qu'ils concernent des changements dans leur environnement d'exécution (modification matérielle, ...), des modifications de leurs composants (ajout, retrait, mise à jour, ...), des comportements internes erronés (composants en erreurs, ...), etc.. Ce domaine a été créé pour faire face à la difficulté grandissante des problèmes rencontrés par l'administration des applications, liée à leur exécution dans des environnements de plus en plus complexes et différents. Une application capable de résoudre par elle-même des problèmes de maintenance permet également de baisser ses coûts d'exploitation, en réduisant le nombre d'interventions humaines nécessaires.

D'après [KC03], une application (ou système) autonomique est un système auto-géré possédant quatre caractéristiques principales :

— Auto-Configuration : le système est capable d'adapter sa configuration, sans intervention humaine, selon son environnement d'exécution et les évolutions de ce dernier ;

— Auto-Réparation : le système est capable de se réparer en cas de dysfonctionnement, c'est-à-dire de réagir face aux erreurs que ses composants produisent ;

— Auto-Optimisation : le système adapte sa configuration afin d'atteindre un mode de fonctionnement optimal (consommation de ressources, temps de calcul, ...), en fonction de son environnement ;

— Auto-Protection : le système peut se protéger de comportements malveillants (accès à des éléments critiques, ...).

[KC03] définit également la notion de *gestionnaire autonomique*, s'appuyant sur une *boucle autonomique*, aussi appelée boucle MAPE-K (*Monitor*, *Analyze*, *Plan*, *Execute*, *Knowledge*), représentée par la figure 6.2. Des capteurs (*sensors*) écoutent les événements concernant l'élément que le gestionnaire autonomique surveille (*managed élément*) ou provenant de son environnement

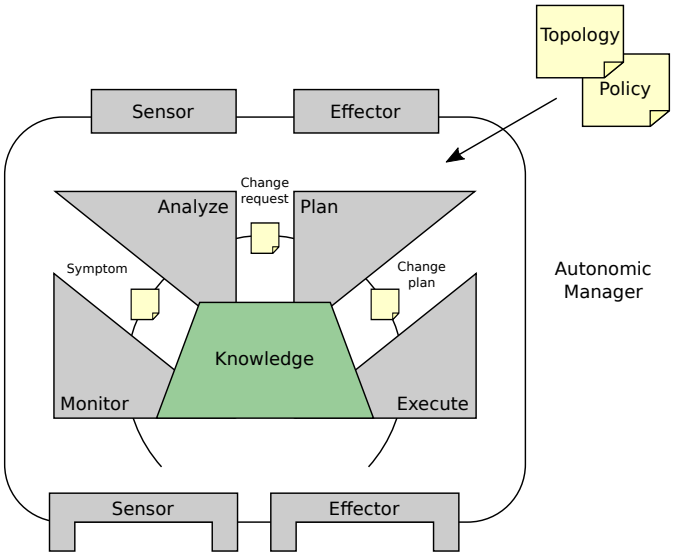

FIGURE 6.2 – Représentation d'une boucle autonomique
Définie par [KC03].

d'exécution. Ces capteurs transmettent ces événements ou sont sondés par la phase de surveillance (*Monitor*) de la boucle. Cette dernière agrège les données et les transforme pour qu'elles correspondent au modèle interne du gestionnaire, puis les enregistre dans l'historique des états de l'élément géré. Cet historique fait partie des connaissances (*Knowledge*) liées à cet élément. La phase d'analyse (*Analyze*) se charge alors de déterminer les actions nécessaires pour répondre aux événements reçus et au nouvel état de l'élément. La phase de planification (*Plan*) définit ensuite l'ordre et l'instant d'exécution des différentes tâches définies lors de la phase d'analyse. Enfin, la phase d'exécution (*Execute*) utilise les actionneurs (*Effectors*) pour appliquer les actions planifiées à l'élément géré et à son environnement. L'historique d'exécution de ces tâches est généralement enregistré dans les connaissances (*Knowledge*) pour que les phases d'analyse et de planification suivantes puissent évaluer les effets de leurs précédentes décisions.

L'autonomie des applications a été classée par [GC03] à travers cinq niveaux, décrits dans le tableau 6.2. Le niveau basique (1) correspond aux applications n'ayant aucune capacité autonomique et qui doivent être gérées par une équipe opérationnelle spécialisée. Les applications qui disposent d'outils d'administration et de surveillance appartiennent au niveau géré (2). Le niveau prédictif (3) regroupe les systèmes capables de recommander les actions que doit effectuer l'équipe opérationnelle, en fonction de son état : les opérateurs ont toujours la maitrise sur les actions à exécuter. Le niveau adaptatif (4) réduit le champ d'action de l'équipe opérationnelle : les applications réagissent automatiquement aux événements qui les concernent, tandis que les opérateurs se chargent de l'optimisation de leurs performances et de leurs qualités de service. Enfin, le niveau autonomique (5) regroupe les applications capables de s'adapter à leur environnement et à

Niveau basique 1	Niveau géré 2	Niveau prédictif 3	Niveau adaptatif 4	Niveau autonomique 5
• Multiples sources de données générées par le système • Besoin de beaucoup de personnel hautement qualifié	• Outils de gestion fournissant des données regroupées • Le personnel analyse et agit	• Le système supervise, corrèle et recommande des actions • Le personnel approuve et initie les actions	• Le système supervise, corrèle et agit • Le personnel gère la performance en terme d'accord de niveau de service	• Des composants intégrés dynamiquement gérés par des règles/politiques du métier • Le personnel se concentre sur les besoins du métier
	• Plus grande connaissance du système • Productivité accrue	• Réduit la nécessité de personnel hautement qualifié • Les décisions sont plus précises et plus rapides	• Applications résilientes et agiles avec un minimum d'intervention du personnel	• Les règles métier dirigent la gestion des technologies de l'information • Applications résilientes et agiles
Contrôle Manuel				Autonomique

TABLE 6.2 – Niveaux autonomiques
Extrait de [GC03], traduit par [Bou08].

leurs événements et d'optimiser leur exploitation. L'équipe opérationnelle n'a plus la tâche d'administrer la configuration de l'application, mais plutôt de formaliser les politiques qu'elle utilisera en fonction des besoins de son domaine métier.

6.5.2 Comparaison avec Cohorte Composer

Cohorte Composer possède certaines des propriétés autonomiques, telles que définies dans la section précédente.

Suivant les définitions de [KC03], Cohorte Composer possède les caractéristiques d'auto-réparation et d'auto-optimisation. L'auto-réparation correspond à la capacité d'instancier à nouveau les composants ayant disparus, en redémarrant des isolats pour les héberger. Ces actions sont effectuées directement par le *Node Composer*, à travers les fonctionnalités de l'isolat moniteur, et ne nécessitent aucune intervention de la part de l'équipe opérationnelle. L'auto-optimisation est visible à travers le compromis que résout le *Node Composer* lors de la distribution de composants, à savoir l'utilisation d'un nombre minimum d'isolats pour héberger l'ensemble des composants d'un nœud.

Cohorte Composer n'est cependant pas capable d'auto-protection. En effet, les services des *Top*, *Node* et *Isolate Composers* n'effectuent aucune vérification sur leurs appelants ni sur les composants qu'ils instancient. Les *composers* sont accessibles depuis n'importe quel composant de l'application, que ce soit pour récupérer des informations sur les compositions actives ou pour en démarrer de nouvelles. L'auto-configuration, qui concerne plutôt Cohorte en général, n'est pas supportée. La plate-forme Cohorte Runtime est seulement en mesure de s'adapter en fonction du système d'exploitation hôte lors de la sélection des outils de surveillance des isolats. De son côté, Cohorte Herald est capable de prendre en compte tous les périphériques réseau présents sur

la machine hôte (virtuelle ou physique) lors du démarrage d'un isolat, afin de dialoguer avec un maximum de pairs, cependant il ne prend pas en charge leur apparition ou disparition.

Les structures de Cohorte Composer et de Cohorte Runtime sont comparables à des boucles autonomiques, telles que représentées par la figure 6.2. Les capteurs (*Sensors*) correspondent au composant *Watcher* de l'isolat moniteur, surveillant l'état des isolats. La phase de surveillance (*Monitor*) est prise en charge par le composant *Monitor* de l'isolat moniteur lors de la perte d'un isolat. La phase d'analyse (*Analyze*) équivaut au traitement des événements par le *Monitor*, le *Node Composer* ou encore l'*Isolate Composer*. Par exemple, durant cette phase, le *Monitor* prépare la configuration de l'isolat à redémarrer (s'il s'agissait d'un isolat configuré), ou le *Node Composer* met à jour ses critères de distribution après avoir été notifié de la perte des composants d'un isolat arrêté. Suivant le même exemple, la phase de planification (*Plan*) correspond à la transmission de l'ordre de démarrage d'un isolat configuré du *Monitor* vers *Forker*, ou au calcul de la nouvelle distribution de composants par le *Node Composer*. Enfin, toujours suivant le même exemple, la phase d'exécution (*Execute*) est traitée, respectivement, par le *Forker*, utilisant les composants *Starters* comme actionneurs (*Effectors*) pour démarrer l'isolat souhaité, ou par l'*Isolate Composer*, exécutant les ordres d'instanciation du *Node Composer* au moyen de ses agents. Le *Monitor* et le *Node Composer* s'appuient sur leurs connaissances (*Knowledge*) de l'application : des configurations d'isolats pour le premier, des compositions et des historiques de distribution pour le second. Ces deux composants utilisent des machines à états finis pour représenter l'état actuel de l'application.

Enfin, selon le classement de [GC03] présenté dans le tableau 6.2, nous considérons que Cohorte Composer est de niveau autonomique (5). En effet, cet outil est capable de répondre seul aux différents événements concernant ses composants, tout en assurant une utilisation optimale des ressources. Bien que l'équipe opérationnelle puisse donner des indications pour optimiser la qualité de service de l'application, *e.g.* en configurant certains isolats, la plate-forme est capable d'atteindre une distribution de composants suffisamment fiable pour obtenir un niveau de qualité équivalent. Le travail de l'équipe opérationnelle consiste à définir une composition, c'est-à-dire la liste des composants nécessaires à l'application pour fournir les fonctionnalités de son domaine métier.

6.6 Conclusion

Ce chapitre a présenté Cohorte Composer. Ce système permet de construire une application à partir d'un ensemble de compositions de composants définis selon des modèles hétérogènes. Dans la description d'une application, une composition est définie par un nom et un composite racine. Un composite est un ensemble hiérarchique de composants et de composites fils. À l'exécution, cette composition est mise à plat, c'est-à-dire qu'elle n'est plus définie que par son nom et par un ensemble de composants. Les noms de ces derniers se seront vus préalablement attribués comme

préfixes les noms de leurs composites parents et le nom de la composition, afin de garantir leur unicité.

Le chargement d'une composition décrite dans un fichier et sa transformation selon le modèle du Composer sont effectués par le *Top Composer*. Celui-ci calcule la répartition nodale des composants de cette composition, puis transmet ses ordres aux *Node Composers*. Les *Node Composers* sont les éléments les plus importants de Cohorte Composer : ils se chargent de calculer la répartition de leurs composants entre différents isolats en fonction des instructions de la composition et de critères de fiabilité. Ils sont notamment capables de gérer leur pan de la composition en autarcie, c'est-à-dire même si le *Top Composer* disparait. Chaque *Node Composer* est exécuté dans l'isolat moniteur de la plate-forme Cohorte Runtime exécutée sur son nœud et peut ainsi lui demander de démarrer les isolats nécessaires au fonctionnement de l'application. Chaque isolat héberge un *Isolate Composer* qui reçoit ses ordres du *Node Composer*. L'*Isolate Composer* a pour tâche d'installer les artefacts contenant les fabriques des composants à instancier. Il utilise ensuite des agents pour interagir avec les plates-formes d'exécution des modèles des composants à instancier.

La table 6.3 compare Cohorte Composer aux canevas présentés dans la section 2.7.

	AIOLOS	**HighFrame**	**SCA**	**Cohorte Composer**
Langages supportés	Java	Java	Selon implémentation (généralement Java)	Java, Python
Modèles supportés	OSGi (SOA)	Composants HighFrame (SOCM)		OSGi (SOA), iPOJO, iPOPO (SOCM)
Artefacts utilisés	Bundle OSGi (fichier JAR)			Bundles OSGi et Pelix (fichier JAR, module Python)
Dépôt d'artefacts	OBR	Composants générés		Dossiers locaux
Communications	R-OSGi	InteropFrame	*Bindings* extensibles, Web-Services	Pelix/Cohorte Remote Services
Description topographique			Fichiers XML (composites)	Fichiers JSON
Support *scaling*	Oui	Oui	Non	Non

TABLE 6.3 – Comparaison avec les canevas de distribution de composants

La majeure partie de Cohorte Composer a été développée en Python. Seuls l'*IsolateComposer* et ses agents ont été également implémentés en Java, afin de gérer les composants iPOJO. La table 6.4 présente le nombre de lignes de code (LOC) qui ont été nécessaires au développement de Cohorte Composer. Comme dans les précédents chapitres, seules les lignes contenant des instructions sont comptées ; les commentaires, la documentation et les lignes vides sont omis.

Faisant partie de la plate-forme Cohorte, le projet Cohorte Composer est un projet libre diffusé

Cohorte Composer	LOC Python	LOC Java	Python+Java
Total Cohorte Composer	2 151	630	2 781
Top Composer	421	0	421
Node Composer	1 280	0	1 280
Isolate Composer	285	630	915
– dont agent iPOPO	*114*	*0*	*114*
– dont agent iPOJO	*0*	*298*	*298*
Système de vote	539	0	539
– dont cœur	*185*	*0*	*185*
– dont vote par approbation	*57*	*0*	*57*

TABLE 6.4 – Lignes de code de Cohorte Composer

sous les termes de la licence Apache Software License 2.0 [8] (ASL 2.0). Il n'est pas disponible comme une bibliothèque indépendante mais il fait partie du livrable Cohorte [9]. Son code source a été intégré à celui de Cohorte Runtime, qui est disponible sur la forge en ligne GitHub [10].

8. `http://www.apache.org/licenses/LICENSE-2.0`
9. `http://cohorte.github.io/downloads/`
10. `https://github.com/isandlaTech/cohorte-runtime`

CHAPITRE 7

EXPÉRIMENTATIONS ET VALIDATIONS

« L'important, c'est de savoir ce qu'il faut observer. »

Edgar ALLAN POE, *Histoires Extraordinaires*, 1856

« Lourd est le parpaing de la réalité sur la tartelette aux fraises de nos illusions. »

BOULET, *Notes, Tome 3 : La viande, c'est la force*, 2009

Sommaire

7.1 Introduction 111
7.2 Agilium NG 112
7.3 Prédis MHI 114
7.4 Robot Nao et OpenHab 115
7.5 Évaluation des performances 116
7.5.1 iPOPO 116
7.5.2 Services Distants 120
7.5.3 Cohorte 126
7.6 Conclusion 129

7.1 Introduction

Dans ce chapitre, nous présentons les différents projets dans lesquels nos travaux ont été utilisés. Nous commençons par la suite logicielle Agilium NG, dont le développement met en œuvre la plate-forme Cohorte. Nous présentons ensuite l'expérience Prédis, qui a été la première à adopter iPOPO. Ce modèle à composants a également été sélectionné dans une démonstration de maison intelligente, utilisant le robot Nao comme interface homme-machine. Enfin, nous terminons ce chapitre par l'évaluation des performances d'iPOPO et de Cohorte.

7.2 Agilium NG

Agilium [1] est une suite logicielle de BPM (*Business Process Management*) développée par la société M1i. Elle permet aux entreprises de modéliser puis de gérer l'exécution et l'évolution de leurs processus métier. Agilium utilise le canevas Cristal [SBM+14] comme moteur d'orchestration, développé pour l'expérience CMS (Solénoïde Compact pour Muons) du LHC (Grand Collisionneur de Hadrons) hébergée par le CERN (Organisation Européenne pour la Recherche Nucléaire). Cette suite logicielle est composée de trois parties : Agilium Factory, une interface de modélisation de processus métier basée sur Eclipse RPC, Agilium Web, une interface graphique pour les utilisateurs finaux, et Agilium Server, le noyau d'exécution de la suite.

Agilium V3

La version actuelle du cœur de cette suite logicielle, Agilium Server v3, est une application Java monolithique qui intègre le moteur d'orchestration, des sous-systèmes spécialisés (serveur REST, ...), ainsi que des modules d'extension. Ces derniers permettent d'ajouter des fonctionnalités spécifiques à certains clients et de les utiliser depuis les activités des processus métier. En cas de défaillance d'une de ces extensions, il est possible que le serveur soit touché et donc s'arrête de manière impromptue. De plus, certaines d'entre elles peuvent être fortement consommatrices en ressources matérielles, au point d'impacter les performances de leur application hôte. Cette version de la suite Agilium a été développée de 2003 à 2011 et représente 124 000 lignes de code Java. Elle a nécessité 200 mois-hommes pour être réalisée.

Agilium NG et Cohorte

Le développement de la nouvelle version de cette suite logicielle, Agilium NG, a été initialement planifié pour utiliser la dernière version du moteur d'orchestration, ainsi que pour adopter une architecture modulaire s'appuyant sur une plate-forme OSGi et sur des composants iPOJO. L'objectif est alors d'obtenir une application extensible dynamiquement. Afin de résoudre la problématique des extensions, la plate-forme Cohorte a été choisie pour isoler des autres composants le cœur de l'application, *i.e.* le moteur d'orchestration et les sous-systèmes critiques. L'isolation des composants développés ou certifiés par la société éditrice d'Agilium, M1i, est définie statiquement, à travers la composition incluse dans la suite logicielle. Les composants développés par des éditeurs tiers sont, quant à eux, isolés selon les règles de fiabilité de Cohorte Composer. Ils peuvent également subir une isolation nodale si l'opérateur d'Agilium Server considère que certains composants abaissent trop les performances globales du système. De plus, l'opérateur de l'application peut également définir lui-même les différents isolats qui doivent héberger ces composants, *e.g.* en utilisant un isolat par fournisseur de module d'extension, afin de mieux déterminer la responsabilité de chacun des acteurs en cas de défaillance.

1. `http://www.agilium.com/`

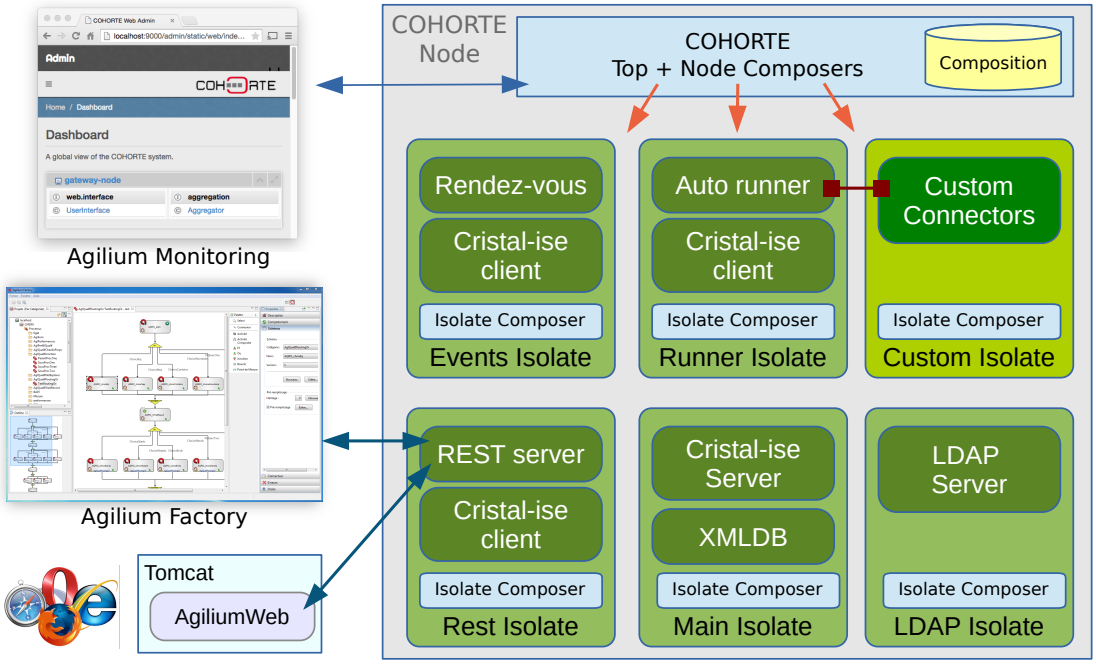

FIGURE 7.1 – Vue globale d'Agilium NG

La figure 7.1 représente la nouvelle architecture du projet. Par défaut, le cœur de l'application est exécuté sur un seul nœud. Le moteur d'orchestration est exécuté avec son gestionnaire de base de données XML dans un isolat spécifique (*Main Isolate*). Le serveur REST, chargé de l'interaction entre le serveur et les utilisateurs est également isolé. Ainsi, toute erreur causée par une entrée utilisateur malformée ou malicieuse n'atteint pas le moteur d'orchestration. L'isolat *Runner* se charge de l'exécution des activités métier automatiques, c'est-à-dire associées à des scripts ou à des connecteurs personnalisés (envoi de mail, génération de documents, ...). Son isolation est due aux mêmes raisons que celle du serveur REST, *i.e.* l'interaction avec des entités tierces non sûres. Les connecteurs sont quant à eux regroupés dans un ou plusieurs isolats appelés *custom*, afin de réduire le risque de propagation de leurs défaillances. Enfin, le serveur LDAP est isolé du fait de sa consommation de ressources mémoire. Il était initialement démarré comme un isolat *exe*. Il a ensuite été transformé en un isolat Java afin d'être surveillé plus efficacement.

Agilium NG a ainsi adopté la plate-forme Cohorte afin de s'assurer que les extensions ajoutées par l'opérateur de l'application n'impactent pas le moteur d'orchestration outre mesure, mais aussi pour garantir un redémarrage automatique des isolats et de leurs composants en cas de défaillance. La capacité d'isolation nodale de Cohorte Composer permet également d'améliorer les performances globales de l'application quand celle-ci doit utiliser des extensions consommatrices de ressources : le moteur d'orchestration a alors suffisamment de ressources pour fonctionner de ma-

nière optimale. Enfin, le support natif du modèle iPOJO par Cohorte Composer a permis d'intégrer très rapidement la plate-forme Cohorte aux développements effectués avant son adoption. Agilium NG est toujours en cours de développement lors de la rédaction de ce manuscrit, cependant l'observation de son code source montre que les développements sont focalisés sur les fonctionnalités métiers du produit, les fonctionnalités techniques étant déjà couvertes par Cohorte.

7.3 Prédis MHI

Prédis est un ensemble de plates-formes dédiées à la recherche et à l'éducation, faisant partie de l'école Grenoble INP–Ense3 (École de l'Énergie, l'Eau et l'Environnement). Ces plates-formes rassemblent plusieurs partenaires académiques et industriels travaillant autour d'axes émergents du génie électrique et de la gestion de l'énergie.

La plate-forme Prédis MHI [2] (Monitoring et Habitat Intelligent), aussi appelée Prédis *Smart Building*, se focalise sur la gestion énergétique de bâtiments et sur les interactions entre des maisons intelligentes (*smart home*) et les grilles de production intelligentes (*smart grid*). Deux laboratoires développent leurs activités de recherche sur cette plate-forme : le G2ELab [3] (Grenoble Génie Électrique) et G-SCOP [4] (Sciences pour la Conception, l'Optimisation et la Production). Les principaux sujets étudiés dans Prédis MHI sont :

— la surveillance multi-capteurs,

— l'analyse des activités de l'utilisateur et de leur impact énergétique,

— la modélisation de chaque équipement physique, prenant en compte les différences entre chaque type d'équipement,

— le développement de stratégies de contrôle optimales.

Nos travaux ont concerné HAL (*Home Abstraction System*), un outil faisant partie du système de contrôle de Prédis MHI. HAL intègre des pilotes qui permettent l'utilisation de différents protocoles pour communiquer avec les périphériques et les capteurs physiques présents dans la salle d'expérimentation. Il agrège les données de ces capteurs et les rend disponibles à travers une interface RESTful, accessible depuis un navigateur Web ou un client lourd. Son objectif est de fournir un accès uniformisé aux informations et aux actionneurs de chaque périphérique.

La version d'origine de HAL a été développée en Python, car la plupart des pilotes utilisés avaient été livrés ou écrits dans ce langage. Elle a été conçue selon une architecture statique : toute modification de la configuration, *i.e.* tout ajout, mise à jour ou retrait de périphérique physique, impliquait le redémarrage complet de cette application de surveillance. De plus, les pilotes des

2. `http://predis.grenoble-inp.fr/smartbuilding/`
3. `http://www.g2elab.grenoble-inp.fr/`
4. `http://www.g-scop.grenoble-inp.fr/`

équipements n'ont pas tous le même degré de qualité et certains s'avéraient moins fiables que d'autres, causant des erreurs de traitement pouvant affecter l'intégrité de l'application HAL. Enfin, un certain nombre de bogues avaient été repérés, notamment au niveau des liaisons entre le serveur HTTP et les pilotes, c'est-à-dire lors du traitement des requêtes REST. D'autres bogues étaient dûs à la non-libération de ressources par les différents pilotes, *e.g.* suite à la fermeture abrupte d'une *socket*.

Nos travaux ont donc consisté en la transformation de l'application HAL en un système modulaire s'appuyant sur iPOPO. En premier lieu, tous les modules Python fournissant des pilotes ont été reconditionnés en bundles Pelix. Pour cela, tout le code d'initialisation du module, qui était auparavant exécuté lors de son import, a été déplacé dans l'activateur du bundle. Cette modification a principalement permis d'ajouter le code inverse, permettant une gestion propre du déchargement du bundle, c'est-à-dire de la libération des ressources utilisées. Ensuite, chacun des pilotes a été transformé en un composant iPOPO et s'est vu ajouté le code nécessaire au traitement de son cycle de vie. Ces deux étapes ont permis de résoudre la majeure partie des bogues liés à la gestion des ressources. Les pilotes fournissent désormais un service correspondant à leurs capacités.

Enfin, le serveur HTTP utilisé par HAL a été remplacé par celui intégré à Pelix. Chaque élément pouvant être appelé à travers l'interface REST de l'application fournit alors un service *servlet*, permettant au serveur de réagir dynamiquement à son apparition et à sa disparition. Du fait de son taux de réutilisation dans d'autres projets et de sa suite de tests unitaires, le service HTTP de Pelix s'est avéré beaucoup plus fiable que le serveur développé spécifiquement pour les besoins de l'application HAL.

En conclusion, la transformation de HAL en un ensemble de composants iPOPO et la réutilisation de services Pelix a permis de résoudre la plupart des bogues que connaissait cette application, tout en lui apportant le dynamisme qui lui faisait défaut. Désormais, la configuration de HAL peut être adaptée directement et sans redémarrage lors de l'ajout ou du retrait de capteurs physiques.

Du fait de l'ajout de code pour libérer les ressources en cas d'arrêt d'un composant ou d'un bundle, le nombre de lignes de code (hors commentaires et lignes vides) n'a que très peu évolué : de 1 562 lignes dans la version d'origine de HAL, le remaniement a abouti à 1 430 lignes dans la version iPOPO.

Ces travaux ont été réalisés avec Shadi ABRAS, sous la supervision de Stéphane PLOIX, Frédéric WURZ, et Benoît DELINCHANT et ont fait l'objet de deux publications [ACP+14, ACD+14].

7.4 Robot Nao et OpenHab

L'objectif de cette expérimentation est de transformer le robot Nao [5], de la société Aldebaran, en une interface homme-machine dans le cadre d'une maison intelligente. Cette maison est gérée

5. `https://www.aldebaran.com/fr/qui-est-nao`

à l'aide de la plate-forme OpenHAB [6], devenue depuis Eclipse Smart Home [7].

Le robot Nao peut exécuter des scripts Python hébergés dans sa mémoire. Ceux-ci ont accès à toutes les fonctionnalités du robot : de la synthèse vocale aux mouvements de ses membres en passant par la reconnaissance d'objets. Nous avons enveloppé toutes ces fonctionnalités dans des composants iPOPO : chacune d'entre elles est alors accessible à travers un service.

À travers le protocole MQTT, le robot reçoit des événements depuis OpenHAB et lui donne des ordres. Ce protocole permet de recevoir des messages et d'en émettre à un grand nombre d'abonnés. Il repose sur un serveur central (courtier, *broker*), pouvant être répliqué. Chaque message est associé à un sujet, à un contenu et à une qualité de service. Cette dernière indique si un message doit être transmis sans garantie de réception (QoS 0), avec garantie de réception (QoS 1) ou avec garantie de réception unique (QoS 2). Les abonnés s'inscrivent auprès du serveur afin de recevoir les messages respectant des filtres simples sur leur sujet.

Cette expérimentation a été développée dans le cadre de la thèse de Wafa JOHAL, avec l'aide de Shadi ABRAS et d'Amr ALYAFI. Elle a été présentée lors des Eclipse IoT Days Grenoble 2014 [8, 9]. Le code source est disponible en ligne sur GitHub [10].

7.5 Évaluation des performances

Cette section présente les différentes mesures de performance effectuées sur iPOPO, Cohorte Herald et sur la plate-forme Cohorte. Elles ont été mises en œuvre afin d'avoir un aperçu de la baisse de performance due à l'adoption de Cohorte, par rapport à une application monolithique.

7.5.1 iPOPO

L'adoption d'un modèle à composants tel qu'iPOPO engendre forcément une baisse des performances, par rapport à une approche objet gérée directement par le développeur. En effet, la plate-forme d'exécution iPOPO doit construire et configurer le conteneur de chaque composant, ainsi que ses *handlers*. La généricité de ce mécanisme d'iPOPO impose aux *handlers* de lier le composant au reste de l'application en recherchant les services qui lui sont nécessaires, ce qui demande du temps. Dans le cadre d'une application suivant l'approche objet, le développeur aurait la capacité de lier directement les différents objets entre eux, sans recherche et donc plus rapidement, mais cela représenterait une charge de travail beaucoup plus importante.

6. http://www.openhab.org/
7. http://www.eclipse.org/smarthome/
8. http://wiki.eclipse.org/Eclipse_IoT_Day_Grenoble_2014
9. https://ipopo.coderxpress.net/wiki/lib/exe/fetch.php?media=contrib:postersamrthomenao.pdf
10. https://github.com/tcalmant/demo-ipopo-nao

Nous avons développé un ensemble de bancs de tests pour mesurer le temps nécessaire à iPOPO pour démarrer différents types de composants. Ces tests consistent à démarrer une plate-forme Pelix, puis à mesurer la durée d'instanciation d'un grand nombre de composants de l'une des quatre fabriques suivantes :

— fabrique vide (`empty`) : cette fabrique est *creuse*, c'est-à-dire qu'elle n'est associée à aucun *handler*. Ses composants sont ainsi les plus rapides à pouvoir être instanciés par iPOPO ;

— fabrique `A` : chaque composant de cette fabrique fournit un service. Chaque instance n'est donc associée qu'à un seul *handler* et ne nécessite pas la recherche de dépendances dans le registre de services.

— fabrique `B` : ses composants requièrent un seul des services fournit par les composants de la fabrique `A`. Ils représentent le comportement de la majorité des composants développés avec iPOPO.

— fabrique `C` : chaque composant de cette fabrique requiert tous les services fournis par tous les composants de la fabrique `A`. Ainsi, chaque composant de cette fabrique consomme l'intégralité du registre de services lors de l'exécution d'un test de ce banc d'essai.

Lors des tests de performance, nous avons mesuré le temps nécessaire pour instancier 1, 500, 1 000, 5 000, 10 000, 50 000 et 100 000 composants de chaque fabrique.

Instanciations à liaisons simples

Fabrique	1	500	1 000	5 000	10 000	50 000	100 000
`empty`	0,103 ms	16,9 ms	35,3 ms	169 ms	340 ms	1,70 s	3,38 s
`A`	0,153 ms	41,1 ms	83,6 ms	429 ms	881 ms	4,48 s	9,06 s
`B`	0,147 ms	46,0 ms	94,3 ms	472 ms	944 ms	4,86 s	9,72 s
`C`	0,157 ms	48,2 ms	99,2 ms	485 ms	978 ms	5,02 s	9,95 s

TABLE 7.1 – Temps d'instanciation de composants des fabriques tests

Les temps des fabriques `B` et `C` ont été mesurés face à une seule instance de composant de la fabrique `A`.

La table 7.1 présente les résultats obtenus, représentés par la figure 7.2. Lors de ces mesures, les composants des fabriques `B` et `C` n'ont été associés qu'à une seule instance de composant de la fabrique `A`. Nous voyons parfaitement que l'évolution des temps nécessaires à l'instanciation de ces composants est linéaire et qu'elle dépend des interactions entre les *handlers* et la plate-forme. En effet, nous remarquons que les composants de la fabrique `A`, qui ne font qu'enregistrer un service, sont instanciés légèrement plus rapidement que ceux des fabriques `B` et `C`, qui recherchent et se lient à un service. De même, les composants de la fabrique `empty` mettent trois fois moins de temps à être instanciés, du fait qu'il n'ont pas de *handler* et donc n'interagissent pas avec la plate-forme.

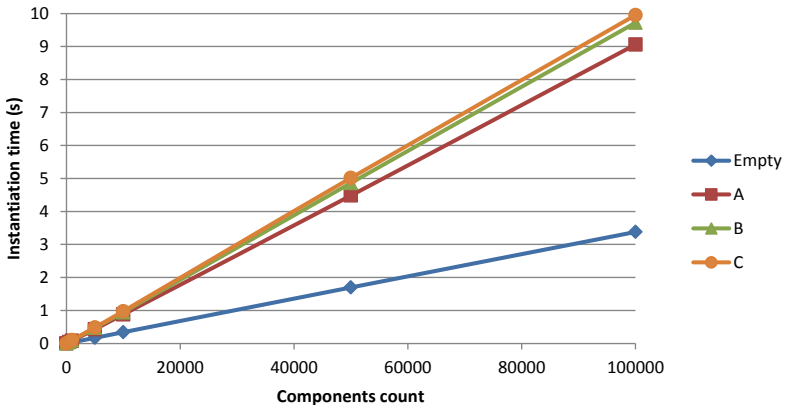

FIGURE 7.2 – Représentation de la table 7.1

Instanciations à grand nombre de liaisons

Nous avons ensuite réalisé les mesures concernant les fabriques `B` et `C` avec 100 puis 1 000 instances de composants de la fabrique `A`, présentées dans la table 7.2. Pour rappel, chaque composant de la fabrique `B` ne consomme qu'un seul service, tandis que ceux de la fabrique `C` consomment chacun tous les services du registre.

Chacun des 100 000 composants de la fabrique `B` récupère le premier service du registre parmi 1 000 ayant pour spécification `A`. Cette recherche ne modifie pas le temps nécessaire à ces instanciations car elle est effectuée à l'aide d'un générateur Python, *i.e.* de manière paresseuse (*lazy)*, dans un registre où les services sont déjà triés dans l'ordre spécifié par OSGi, *i.e.* par priorité (*ranking*) puis par identifiant de service. Les faibles écarts de temps visibles sont dus à l'indéterminisme des structures de données utilisées provisoirement pendant le démarrage d'un composant et lors de la recherche de services. En effet, celles-ci s'appuient sur des signatures d'objets (*hashes*) pour ordonner leur contenu, or Python ajoute un grain de sel aléatoire dans ces signatures et modifie donc l'ordre de ces contenus à chaque nouvelle exécution.

Fabrique	Instances A	1	500	1 000	5 000	10 000	50 000	100 000
B	1	0,147 ms	46,0 ms	94,3 ms	472 ms	944 ms	4,86 s	9,72 s
	100	0,165 ms	46,9 ms	94,5 ms	482 ms	964 ms	4,81 s	9,81 s
	1 000	0,191 ms	46,4 ms	95,9 ms	481 ms	968 ms	4,85 s	9,69 s
C	1	0,157 ms	48,2 ms	99,2 ms	485 ms	978 ms	5,02 s	9,95 s
	100	1,97 ms	1,41 s	3,44 s	40,1 s	147 s	4 150 s	–
	1000	21,1 ms	40,7 s	146 s	4 220 s	–	–	–

TABLE 7.2 – Temps d'instanciation de composants des fabriques `B` et `C`

L'instanciation des composants de la fabrique `C` prend beaucoup plus de temps, car chacune de

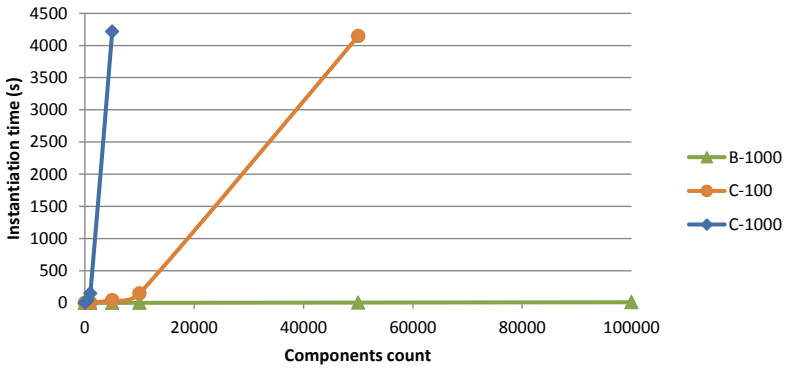

FIGURE 7.3 – Temps d'instanciation des composants `B` et `C` pour 100 et 1 000 composants `A`

ces instances doit se lier aux 100 services fournis par les composants de la fabrique `A`. iPOPO nécessite ainsi près de 4 200 secondes (soit 1 heure et 10 minutes) pour mettre en place les 5 000 000 de liaisons entre 50 000 composants `C` et 100 services `A`, ou entre 5 000 composants `C` et 1 000 services `A`. Cela revient à 0,84 ms en moyenne par liaison, ce qui reste proche de ce qui a été observé lors des mesures présentées dans la table 7.2. La mesure du temps de démarrage de 100 000 composants `C` pour 100 composants `A` a été annulée du fait de sa durée minimale estimée à 9 150 secondes (soit environ 2 heures et 33 minutes). De même, les mesures avec 1 000 composants `A` ont été stoppées dès qu'elles ont nécessité plus d'une heure d'exécution.

Synthèse

Ces mesures ont permis de montrer que le temps de démarrage d'une application s'appuyant sur un très grand nombre de composants iPOPO n'est pas anodin. Ce ralentissement est notamment dû aux structures de données utilisées par les *handlers* et par la plate-forme Pelix pour conserver des informations sur les fournisseurs et les consommateurs de services. Des optimisations avaient été mises en place lors des travaux effectués sur la plate-forme Prédis MHI, pendant lesquels ce protocole de test avait été défini. Nous citerons, par exemple, le parcours paresseux du registre de services ou encore le remplacement de listes internes par des structures de données. Le test majeur consistait alors à instancier 50 000 composants fournissant un service `Svc.Test` et requérant tous les services enregistrés avec cette spécification. Ainsi, chaque nouveau composant fournit un service qui est alors consommé par ceux instanciés avant lui. Tandis que la version d'iPOPO de cette période prenait initialement 3 heures et 30 minutes pour instancier ces composants, les optimisations apportées ont permis de descendre jusqu'à seulement 4 minutes et 30 secondes, soit un démarrage 46 fois plus rapide.

Les appels que les composants effectuent entre eux sont directs, c'est-à-dire sans contrôle de la plate-forme. Pelix et iPOPO n'ont ainsi plus d'impact sur les performances de l'application au cours de son exécution, une fois que tous ses composants ont été instanciés.

7.5.2 Services Distants

La distribution des composants d'une application entre différents isolats impose l'utilisation de services distants pour les lier entre eux. Les étapes de transformation de chaque appel distant vers puis depuis un protocole de transport sont coûteuses et réduisent les performances globales de l'application. Nous avons développé un banc d'essai afin de mesurer le temps nécessaire à un appel de service à distance, en fonction des différentes implémentations de transport développées dans le cadre de cette thèse et par rapport à un appel local direct.

Le service de test utilisé dispose de trois méthodes :

— `dummy()` : cette méthode n'exécute aucun code et renvoie `None`. Elle permet d'estimer le temps nécessaire à la transmission d'un ordre d'appel, à la recherche de la méthode à exécuter et à la transmission de la valeur de retour.

— `single_arg(value)` : cette méthode retourne immédiatement l'horodatage de son exécution. Elle permet ainsi de calculer le temps requis pour transmettre différents types d'arguments.

— `var_args(*args)` : cette méthode fonctionne comme `single_arg`, mais supporte un nombre variable d'arguments. Elle permet de déterminer l'influence du nombre d'arguments lors des appels distants.

Les tests ont été effectués sur une machine disposant d'un processeur Intel® Core™ i5 double-cœur cadencé à 2,50 GHz et de 8 Go de RAM. Les essais ont été réalisés sur la même machine : un pair Herald fournissait le service tandis qu'un autre exécutait les bancs d'essais. Les communications HTTP passaient ainsi par l'interface locale. Le serveur XMPP utilisé pour ces tests était celui de la société isandlaTech, hébergé par Online.net à Paris. Lors des essais, le temps nécessaire à un paquet pour atteindre le serveur et revenir à l'émetteur a été mesuré à l'aide de la commande `ping` et était de 55 ms.

Enfin, les tableaux présentés ci-dessous n'indiquent pas les temps mesurés lors de l'exécution des tests par appel local direct. Ces temps sont de l'ordre de la nanoseconde, soit trois ordres de grandeurs inférieurs à ceux des tests par appel à distance.

Durée d'un aller-retour

Le premier test effectué compare le temps nécessaire pour appeler une méthode d'un service distant et pour recevoir une réponse. Nous utilisons pour cela la méthode `dummy()`, qui ne prend aucun paramètre et n'exécute aucun code. Ainsi, les mesures correspondent à la durée de la transmission de la requête d'appel, de l'analyse de cette requête et de la recherche de la méthode à appeler côté serveur, et de la transmission de la réponse.

La table 7.3 présente les mesures relevées. Chaque valeur de la deuxième colonne correspond au temps nécessaire à un appel et à la réception de la réponse. La troisième colonne indique le

Mode appel	`dummy()`	Temps intermédiaire estimé
Pelix, *via* HTTP		
XML-RPC	2,09 ms	1,05 ms
JSON-RPC	2,14 ms	1,07 ms
Jabsorb-RPC	2,76 ms	1,38 ms
Herald, *via* HTTP		
JSON-RPC	9,73 ms	4,87 ms
Jabsorb-RPC	13,2 ms	6,59 ms
Herald, *via* XMPP		
JSON-RPC	201 ms	100 ms
Jabsorb-RPC	210 ms	105 ms

TABLE 7.3 – Durées moyennes des appels distants à la méthode `dummy`

temps estimé pour la transmission de la requête d'appel, soit la moitié du temps d'un aller-retour. Elle permet une meilleure appréciation des différents temps mesurés par la suite.

Le coût des modifications apportées par Jabsorb-RPC par rapport à JSON-RPC, c'est-à-dire de l'ajout des informations sur le type Java des objets transmis, est visible. En effet, l'implémentation de Jabsorb-RPC en Python consiste à utiliser l'implémentation de JSON-RPC et à modifier le format des requêtes lors de leur émission, et celui des réponses lors de leur réception.

La différence de temps entre Pelix et Herald HTTP est due à leur méthode d'émission des réponses. Pelix suit le modèle requête–réponse de HTTP : le résultat de l'appel est transmis dans la réponse à la requête d'appel. À l'inverse, Herald HTTP répond immédiatement à l'émetteur qu'il a reçu la requête, puis envoie le résultat de l'appel par une nouvelle requête vers le serveur HTTP de l'émetteur.

De même, l'utilisation de XMPP augmente fortement la durée d'un appel. La transmission du message de requête de l'émetteur vers le récepteur prend environ 55 ms, d'après la commande `ping`. Cette valeur n'est pas fixe et a évolué entre 48 ms et 59 ms au cours des tests. À cette durée, il faut ajouter le temps nécessaire au serveur XMPP pour router le message. L'émission de la réponse s'effectue dans les mêmes conditions, ce qui fait un total théorique d'environ 110 ms passées à transmettre les messages. La durée d'un appel distant sur le transport XMPP est ainsi deux ordres de grandeur plus longue que sur le transport HTTP.

Ces résultats montrent que l'utilisation du protocole XMPP doit être limitée aux appels de services géographiquement distants et inaccessibles depuis HTTP, *e.g.* s'il sont protégés derrière un pare-feu. De même, ils permettent de voir que l'approche requête–requête HTTP de Herald est en moyenne 4 fois plus lente que l'approche standard requête–réponse HTTP utilisée par Pelix.

Transmission d'un argument

Le deuxième banc d'essai que nous avons développé mesure le temps nécessaire à la transmission de différents types et différents nombres d'arguments.

La première phase de ce test consiste à appeler la méthode `single_arg(value)`, acceptant un paramètre et retournant l'horodatage de son exécution. De cette manière, nous pouvons estimer le temps passé à transmettre la requête au serveur en fonction du type de paramètre utilisé.

Mode appel	`None`	`int`	`float`	`str`	`list`	`dict`
Pelix, *via* HTTP						
XML-RPC	0,99 ms	0,99 ms	0,99 ms	0,99 ms	1,00 ms	2,00 ms
JSON-RPC	0,99 ms	0,99 ms	0,99 ms	0,99 ms	0,99 ms	1,00 ms
Jabsorb-RPC	1,00 ms	0,99 ms	0,99 ms	0,99 ms	1,99 ms	2,99 ms
Herald, *via* HTTP						
JSON-RPC	4,00 ms	4,99 ms	4,00 ms	4,99 ms	5,00 ms	5,00 ms
Jabsorb-RPC	4,00 ms	5,00 ms	4,00 ms	5,00 ms	7,99 ms	8,00 ms
Herald, *via* XMPP						
JSON-RPC	124 ms	122 ms	125 ms	126 ms	123 ms	115 ms
Jabsorb-RPC	163 ms	165 ms	168 ms	171 ms	188 ms	124 ms

TABLE 7.4 – Durées moyennes des appels distants à la méthode `single_arg`

La table 7.4 présente les mesures effectuées lors de cette première phase. Celles-ci montrent que les arguments de type primitif (`None`, `int`, `float` ou `str`) nécessitent tous autant de temps pour être transmis. Les types complexes, *i.e.* `list` et `dict`, sont des types primitifs en JSON, ce qui explique qu'il n'y ait pas de différence avec les autres types dans le cas de JSON-RPC. Les listes et dictionnaires transmis lors de ces essais contiennent chacun 100 éléments.

Les listes sont traitées de manière générique par XML-RPC : chaque entrée de la liste correspond à un élément fils `value` d'un élément XML `array`. Leur transformation ne correspond qu'à un parcours de liste et est donc peu coûteuse. Chaque dictionnaire est converti en un élément XML `struct`, contenant la liste de ses éléments (`member`). Chacun de ces éléments possède deux enfants : une clé du dictionnaire(`name`) et la valeur associée (*value*). La transformation des dictionnaires correspond donc à un parcours de leurs clés puis à la recherche de la valeur associée, ce qui explique qu'elle prenne plus de temps que la transformation des listes.

Enfin, le tableau met en avant le traitement effectué sur les listes et les dictionnaires par Jabsorb-RPC. En effet, le format Jabsorb nécessite la transformation des listes en un objet JSON contenant une entrée `javaClass` indiquant un type Java de liste (`java.util.ArrayList`) et une entrée `list` contenant la liste elle-même. Un dictionnaire est traité de façon équivalente : l'entrée `javaClass` a pour valeur `java.util.HashMap` et son contenu est déplacé dans l'entrée `map`. Chaque élément des listes et des dictionnaires est analysé, afin d'appliquer récur-

sivement ces modifications aux autres listes et dictionnaires. Ce traitement a un coût temporel, visible dans les mesures présentées. De plus, les modifications apportées augmentent la taille des messages transmis, ce qui se traduit par un traitement plus long par le serveur central dans le cas du protocole XMPP.

Protocole	**Aucun**	`None`	`int`	`float`	`str`	`list`	`tuple`	`dict`
Pelix, *via* HTTP								
XML-RPC	110	153	164	170	265	3067	3067	6710
JSON-RPC	94	117	118	118	215	503	503	1257
Jabsorb-RPC	94	117	118	118	215	549	203	1300
Herald, *via* HTTP								
JSON-RPC	459	484	485	485	584	870	870	1824
Jabsorb-RPC	462	487	488	488	587	925	873	1876
Herald, *via* XMPP								
JSON-RPC	778	813	814	814	923	1199	119	3153
Jabsorb-RPC	784	819	820	820	929	1287	1205	3238

TABLE 7.5 – Taille en octets du corps des requêtes d'appel distant selon le type d'argument

La table 7.5 indique la taille, en octets, du corps de chaque requête HTTP ou message XMPP émis lors d'un appel distant, en fonction de l'argument à transmettre. Nous observons que le résultat de la transformation des listes, des tuples et des dictionnaires par le transport XML-RPC augmente de manière importante la taille des requêtes transmises, comparée à celle des requêtes transmises en JSON. Cette augmentation est due à l'utilisation de balises XML pour définir les valeurs de ces collections.

La transformation opérée par Jabsorb-RPC est visible sur les listes et les dictionnaires. Pour mémoire, cette transformation consiste à insérer un champ `javaClass` indiquant le type Java à utiliser pour charger ces données. Jabsorb-RPC ne modifie pas les objets de type `tuple` car ils sont convertis en tableaux Java plutôt qu'en listes.

Dans le cas de Herald, chaque appel distant correspond à l'émission d'un message Herald. Ainsi, un appel JSON-RPC ou Jabsorb-RPC est le même que celui émis par Pelix, mais il est transmis comme le contenu d'un message Herald, avec l'ensemble de métatonnées et d'en-têtes décrits dans la section 4.2.2, page 69.

Lors de l'utilisation du protocole XMPP, le contenu des messages est traité afin de respecter la norme XML. Ainsi les caractères spéciaux contenus dans la requête sont convertis au format XML, *e.g.* le caractère `"` utilisé pour délimiter les chaînes de caractères en JSON devient `"`. Cette transformation cause l'augmentation de la taille des données transmises visible dans la table 7.5.

Transmission de plusieurs arguments

La seconde phase de ce test consiste à appeler la méthode `var_arg(value)`, acceptant un nombre variable de paramètres et retournant l'horodatage de son exécution. De cette manière, nous pouvons estimer le temps passé à transmettre la requête au serveur en fonction du type et du nombre de paramètres utilisés.

	`int`		`list`		`dict`	
Mode appel	**2**	**10**	**2**	**10**	**2**	**10**
Pelix, *via* HTTP						
XML-RPC	1,00 ms	1,00 ms	1,00 ms	7,00 ms	4,00 ms	15,0 ms
JSON-RPC	1,00 ms	1,00 ms	2,00 ms	3,02 ms	2,00 ms	4,00 ms
Jabsorb-RPC	1,00 ms	1,00 ms	4,00 ms	13,0 ms	5,00 ms	21,0 ms
Herald, *via* HTTP						
JSON-RPC	4,99 ms	5,00 ms	6,00 ms	7,98 ms	6,00 ms	8,98 ms
Jabsorb-RPC	4,99 ms	5,00 ms	7,00 ms	18,0 ms	9,00 ms	27,0 ms
Herald, *via* XMPP						
JSON-RPC	126 ms	128 ms	120 ms	128 ms	127 ms	327 ms
Jabsorb-RPC	127 ms	130 ms	128 ms	157 ms	134 ms	394 ms

TABLE 7.6 – Durées moyennes des appels distants à la méthode `var_args`

La table 7.6 présente les mesures obtenues lors de la transmission de 2 et 10 arguments de type `int`, `list` et `dict`. Des mesures ont été effectuées pour les autres types primitifs mais leurs valeurs étaient proches de celles de la table 7.4 et n'étaient donc pas significatives. Les entiers transmis lors des tests étaient compris entre 2^{2048} et 2^{4096}, mais leur taille ne semble pas avoir d'impact sur les performances. Les listes et dictionnaires transmis lors de ces essais contiennent chacun 100 éléments. Comme dans la première phase, JSON-RPC apparait le plus efficace étant donné qu'il n'effectue aucun traitement sur ces types de données. À l'inverse, la transformation nécessaire à Jabsorb-RPC est bien visible, notamment en ce qui concerne les dictionnaires. Enfin, la transmission de 10 dictionnaires de 100 éléments augmente significativement la taille des messages émis, ce qui explique le temps nécessaire à leur transmission *via* un serveur XMPP.

Appels simultanés

Le dernier test que nous avons développé mesure le coût des appels simultanés à un même service distant. Plusieurs consommateurs vont ainsi appeler simultanément la méthode `dummy()`. Chaque consommateur est dans un fil d'exécution (*thread*) particulier et effectue ses appels au service distant sans interruption. Le démarrage des appels est synchronisé, afin que tous les consommateurs soient initialisés et prêts lors du lancement du test. Ce test consiste à appeler 10 000 fois la méthode `dummy()` en utilisant différents nombres de *threads*. Ainsi, nous avons utilisé 10

threads effectuant 1 000 appels, 50 *threads* effectuant 200 appels et enfin 100 *threads* effectuant 100 appels. Les résultats de ces tests sont les moyennes des meilleurs temps de chaque *thread*.

Mode appel	10 threads	50 threads	100 threads
Pelix, *via* HTTP			
XML-RPC	12,6 ms	63,8 ms	132 ms
JSON-RPC	13,0 ms	67,4 ms	122 ms
Jabsorb-RPC	14,3 ms	72,3 ms	132 ms
Herald, *via* HTTP			
JSON-RPC	70,0 ms	432 ms	972 ms
Jabsorb-RPC	82,6 ms	449 ms	790 ms
Herald, *via* XMPP			
JSON-RPC	265 ms	1,28 s	2,28 s
Jabsorb-RPC	276 ms	1,24 s	2,33 s

TABLE 7.7 – Durées moyennes par appel lors d'appels distants simultanés à la méthode `dummy`

La table 7.7 présente le temps moyen nécessaire à X threads pour effectuer un appel à la méthode `dummy()` et pour recevoir sa réponse. L'ordre de grandeur de différence entre 10 et 100 consommateurs montre qu'il est préférable d'utiliser une queue d'appel à un service distant plutôt que d'effectuer ces appels simultanément. De plus, 5% des appels effectués *via* HTTP ont échoués, du fait du rejet de connexions des consommateurs par le serveur. En effet, chaque appel correspond à une nouvelle requête HTTP et, dans le cas de Pelix, à une nouvelle connexion au serveur. Herald utilise un *pool* de connexions, c'est-à-dire que tous les consommateurs réutilisent les mêmes connexions (*sockets*) pour émettre leurs requêtes. Cependant, la technique requête–requête utilisée par Herald nécessite deux fois plus de connexions que Pelix et peut entrainer la saturation du serveur HTTP du pair émetteur, *i.e.* la perte de messages de réponse.

La liaison XMPP se fait quant à elle *via* une seule connexion : chaque message émis par les consommateurs est placé dans une file d'attente puis transmis au serveur. Le serveur central utilise également des files d'attente pour transmettre séquentiellement chaque message vers son destinataire. Le traitement et la protection contre les modifications simultanées de ces files d'attente sont visiblement coûteux.

Synthèse

Ces tests ont permis de mesurer le coût d'un appel distant dans différents contextes. Ces valeurs montrent que l'utilisation de la plate-forme Cohorte n'est pas neutre. Les appels entre composants isolés d'une application Cohorte prennent au moins 5 ms, au lieu de moins de 1 ns pour un appel local direct, soit trois ordres de grandeur de plus. Enfin, les mesures concernant le protocole XMPP montrent qu'il est préférable de réserver ce protocole pour la liaison de services éloignés

géographiquement ou protégés derrière un pare-feu. L'utilisation de ce protocole pour l'intégralité d'une application causerait une baisse de performance beaucoup trop importante.

Enfin, ces mesures ont conduit à définir des règles de bonnes pratiques pour les opérateurs d'applications Cohorte, mais aussi pour les développeurs. Les opérateurs doivent ainsi regrouper les composants ayant de nombreuses interactions, afin de limiter l'usage des services distants. Cohorte Composer a pour objectif de limiter le nombre d'isolats mais ne prend pas en compte les liaisons entre composants. Le calcul d'une distribution peut donc mener à une forte baisse des performances, du fait de liaisons distantes trop coûteuses. Les développeurs doivent quant à eux limiter la concurrence de leurs appels vers le même isolat afin d'éviter de saturer les serveurs, que ce soit le serveur central XMPP ou le serveur HTTP de l'isolat cible.

7.5.3 Cohorte

La généricité de la plate-forme Cohorte a un impact sur les performances de démarrage d'une application. Nous avons inséré des sondes à différentes étapes clés du démarrage d'une application s'appuyant sur Cohorte afin de mesurer cet impact. Pour cela, nous avons défini quatre compositions de 1, 10, 50 et 100 composants iPOPO. Ces composants sont tous de la même fabrique et n'utilisent aucun *handlers*, afin de ne pas avoir à prendre en compte le temps nécessaire à iPOPO pour gérer d'éventuelles liaisons. Lors de ces mesures, les communications entre les isolats étaient assurées par Herald HTTP. Les isolats étaient des isolats Python démarrés par *bootsrap* (`pelix_inner`).

Démarrage de la plate-forme

	Temps moyen	**Écart-Type**
Moniteur		
Démarrage	4,20 s	0,264 s
Isolat		
Démarrage	577 ms	41,3 ms
Puis découverte par *Node Composer*	52,1 ms	5,08 ms

TABLE 7.8 – Temps moyen de démarrage des isolats

La table 7.8 présente la moyenne des temps nécessaires aux démarrages de l'isolat moniteur et d'un isolat Python (*i.e.* de type `pelix_inner`). Ces étapes ne dépendent pas du nombre de composants à instancier, mais sont sensibles à l'état de leur environnement. En effet, l'opération la plus coûteuse lors du démarrage de l'isolat moniteur est l'analyse de tous les artefacts présents dans ses chemins de recherche, dont font parties les dépôts d'artefacts de l'application et le `PYTHONPATH`, contenant la bibliothèque standard Python. Elle nécessite la lecture et l'analyse

d'au moins 1 000 fichiers et dépend donc à la fois des performances du matériel et de celles du système d'exploitation hôte. Le démarrage d'un isolat par *bootstrap* dépend quant à lui du temps nécessaire à la récupération de sa configuration *via* une requête HTTP vers l'isolat moniteur, ainsi qu'au chargement de ses *bundles*.

La ligne "*découverte par Node Composer*" indique le temps passé entre la validation du composant *Isolate Composer* et sa consommation par le *Node Composer*. La validation de l'*Isolate Composer* marque la fin du démarrage de l'isolat, *i.e.* tous les composants spécifiques à Cohorte ont été validés. Sa consommation par le *Node Composer* assure quant à elle que la liaison entre cet isolat et l'isolat moniteur est fonctionnelle.

Instanciation des composants

	1	**10**	**50**	**100**
Instanciation des composants	14 ms	36 ms	120 ms	210 ms
Démarrage isolat complet	630 ms	710 ms	730 ms	830 ms
Démarrage application	11,0 s	11,2 s	11,1 s	11,8 s

TABLE 7.9 – Temps de démarrage d'une application en fonction du nombre de composants

La table 7.9 présente les mesures de temps de démarrage de l'isolat initial contenant tous les composants. Lors du calcul de la première distribution, si une composition ne définit aucune contrainte sur l'hébergement des composants, le *Node Composer* regroupera tous ses composants dans le même isolat. Cette distribution est en effet la seule respectant l'objectif de démarrer un nombre minimal d'isolats. La première ligne de cette table indique la durée moyenne de l'instanciation des composants à partir de l'instant où le *Node Composer* a détecté l'*Isolate Composer*. Cette durée correspond à deux phases : (*i*) à la transmission de l'ordre d'instanciation du *Node* vers l'*Isolate Composer* puis de l'*Isolate Composer* vers ses agents, et (*ii*) aux instanciations des composants demandés. Les temps indiqués sont plus lents que ceux mesurés avec iPOPO dans la section 7.5.1 car les agents effectuent des opérations supplémentaires pour chaque instance de composant, *e.g.* définir des propriétés supplémentaires ou encore s'inscrire aux événements de l'instance.

Le temps de démarrage de l'isolat complet dépend principalement des accès disques nécessaires au chargement de ses *bundles*, mais le nombre de composants à instancier a également un impact visible. Enfin, la dernière de ligne de cette table présente la durée du démarrage de l'application complète, c'est-à-dire le temps écoulé entre l'exécution de l'isolat moniteur et l'instanciation du dernier composant. L'impact du nombre de composants à instancier est toujours constaté, mais il ne représente qu'une faible proportion (environ 5%) de la durée de démarrage totale, notamment par rapport au temps de démarrage de l'isolat moniteur présenté par la table 7.8.

Perte d'un isolat

Lors de chaque test, nous avons forcé l'arrêt de l'isolat hébergeant les composants. Pour cela, nous avons développé un script qui émet un signal d'interruption (`SIGINT`) vers le processus de cet isolat et qui conserve l'horodatage de cette opération. Nous avons ensuite utilisé les sondes mises en place pour les mesures précédentes afin de déterminer le temps nécessaire à la réapparition de tous les composants disparus.

	1	**10**	**50**	**100**
Détection et redistribution	5,24 s	5,31 s	5,44 s	5,51 s
Démarrage des isolats	0,8 s	1,8 s	1,8 s	1,8 s
Durée avant retour complet	6,07 s	7,06 s	7,23 s	7,32 s

TABLE 7.10 – Temps de redémarrage d'une composition en fonction du nombre de composants

La table 7.10 présente les mesures relevées. La première ligne correspond au temps nécessaire à l'isolat moniteur pour (*i*) détecter la disparition de l'isolat puis pour (*ii*) calculer une nouvelle distribution des composants perdus et enfin pour (*iii*) ordonner le démarrage de nouveaux isolats. Dans le cas de la composition ne définissant qu'un seul composant, seul un isolat est redémarré. La deuxième ligne indique le temps de démarrage des isolats, c'est-à-dire le temps entre l'ordre de démarrage du premier isolat et l'instanciation du dernier *Isolate Composer*. Enfin, la troisième ligne indique la durée entre l'arrêt de l'isolat et la réapparition de tous ses services, quand tous les composants perdus ont été réinstanciés.

Suite à la perte de l'isolat initial, le critère s'appuyant sur l'historique des crashs (présenté dans la section 6.4.1, page 99) interdit au *Node Composer* de regrouper tous les composants dans le même isolat. Ainsi, la nouvelle distribution isole l'un de ces composants dans un premier isolat spécifique et place tous les autres dans un second isolat. Les liaisons par services distants sont alors limitées à celles concernant le composant isolé, et le nombre d'isolats démarrés reste minimal. Si l'isolat contenant un seul composant est à nouveau arrêté, il est redémarré tel quel. Si l'autre isolat est arrêté, l'un de ses composants sera placé dans un nouvel isolat spécifique, tandis que tous les autres sont instanciés dans l'isolat resté actif. Nous conservons ainsi une distribution sur seulement deux isolats, et l'instanciation de la majeure partie des composants n'a pas à attendre le démarrage de l'isolat qui doit les héberger.

Enfin, il est important de noter que le démarrage des isolats est séquentiel : l'isolat moniteur attend que le script de *bootstrap* d'un isolat ait terminé son exécution, *i.e.* que l'*Isolate Composer* soit instancié, avant de démarrer l'isolat suivant. Ce comportement explique la différence de temps entre le démarrage d'un seul isolat (composition à 1 composant) et celui de deux isolats.

Synthèse

Les mesures effectuées ont montré que le temps de démarrage d'une application Cohorte dépend principalement du nombre d'artefacts auxquels a accès l'application. En effet, ces artefacts sont analysés par l'isolat moniteur afin de déterminer leurs inter-dépendances, mais aussi pour associer chaque fabrique de composant à l'artefact la fournissant. Cependant, cette opération n'est effectuée qu'une seule fois, lors du démarrage de l'application, ce qui limite son impact sur les performances globales d'une application à longue durée d'exécution. Un mécanisme de cache pourrait être implémenté afin de réduire la durée de cette opération.

Les opérations des agents de l'*Isolate Composer* (définition de propriétés, inscription aux événements des instances, ...) ont un impact visible sur les performances de l'application. Ainsi, l'instanciation d'un ensemble de composants nécessite plus de temps avec ce mécanisme qu'une instanciation spécifique, telle que mesurée avec iPOPO dans la section 7.5.1. Cette différence reste cependant négligeable car les durées d'instanciation sont du même ordre de grandeur.

Enfin, nous avons mesuré le temps nécessaire pour qu'une application fournisse à nouveau l'intégralité de ses services après que l'un de ses isolats ait disparu. La majeure partie de ce temps est passé dans la découverte de la perte de l'isolat et dans le calcul de la nouvelle distribution des composants perdus. Le cas présenté est l'un des plus lents : l'intégralité des composants est perdue et deux isolats doivent être démarrés pour héberger la nouvelle distribution. Dans le cas général, seul un isolat doit être démarré, et une partie des composants disparus sont instanciés sur un isolat déjà présent. Enfin, nous avons observé que la durée nécessaire à la réapparition de l'intégralité des services est inférieure à celle de démarrage de l'application et dépend plus du nombre d'isolats à démarrer que du nombre de composants à instancier.

7.6 Conclusion

Ce chapitre a présenté les différentes utilisations des plates-formes iPOPO et Cohorte.

iPOPO a principalement été testé dans le cadre des maisons intelligentes, du fait de l'utilisation du langage Python dans la plupart des expérimentations. Son adoption a, par exemple, permis de réduire drastiquement le nombre de bogues présents dans l'application HAL de la plate-forme Prédis MHI, en raison de l'apport d'un cycle de vie à ses différents composants et de la réutilisation de services robustes.

Cohorte sert désormais de socle à une suite logicielle commerciale, Agilium NG. Son mécanisme d'isolation permet de protéger le cœur de l'application contre les défaillances des composants tiers, voire des sous-systèmes non critiques.

En dehors de l'expérience Prédis MHI, il ne nous est pas possible de fournir de comparatifs de performances entre une version utilisant Cohorte et une version témoin. En effet, dans le cas d'Agilium NG, la réécriture causée par l'adoption d'OSGi et l'utilisation d'un nouveau moteur

d'orchestration fausserait grandement ces informations.

Nous avons ensuite présenté les mesures effectuées pour estimer l'impact de l'utilisation de Cohorte sur les performances d'une application. Les résultats concernant iPOPO sont encourageant dans le cadre d'instances de la plate-forme ne devant pas gérer plus de 100 000 liaisons. Au delà, le temps d'instanciation dépasse 10 secondes et a donc un impact non négligeable sur l'application.

Les mesures concernant les services distants ont montré les limites de Cohorte Herald. L'utilisation de son transport XMPP doit ainsi être limitée à des cas bien précis, *e.g.* pour communiquer avec des pairs qui ne sont pas accessibles directement. Les performances de son transport HTTP sont tolérables, mais supportent mal la charge, *i.e.* les appels simultanés. Nous avons également pu évaluer le coût des transformations de Jabsorb-RPC, nécessaires pour autoriser les communications entre des composants Python et Java. Bien qu'elles ne modifient pas l'ordre de grandeur de la durée d'un appel distant, le ralentissement qu'elles causent est perceptible.

Enfin, nous avons évalué le temps de démarrage d'une application Cohorte ainsi que le temps de redémarrage de composants perdus. Une application Cohorte démarre en un peu plus de 10 secondes, les 5 premières secondes étant dues à l'analyse des artefacts par l'isolat moniteur. Le redémarrage des composants d'un isolat disparu dure environ 7 secondes. Bien que ces performances ne soient pas satisfaisantes, elles restent plus rapides qu'une opération de redémarrage manuelle. De plus, bien que le calcul de la distribution des composants perdus ralentisse leur réapparition, il permet de mieux isoler les composants afin qu'une majorité d'entre eux soit épargnée des erreurs pouvant apparaitre plus tard.

CHAPITRE 8

CONCLUSION ET PERSPECTIVES

> *Il n'y a rien de noble à être supérieur à vos semblables.*
> *La vraie noblesse est d'être supérieur à celui que vous avez été auparavant.*

Ernest HEMINGWAY

Sommaire

8.1 Synthèse 131
8.1.1 Rappel de la problématique 131
8.1.2 Approche et exigences 132
8.1.3 Contributions 133
8.2 Perspectives 136
8.2.1 Travaux en cours 136
8.2.2 Travaux envisagés 138

8.1 Synthèse

8.1.1 Rappel de la problématique

Depuis les années quatre-vingt-dix, le Génie Logiciel adopte de plus en plus les méthodes agiles de gestion de projets au détriment des méthodes fortement planifiées empruntées à l'industrie lourde. Tandis que ces dernières imposaient des cycles de développement rigides, le principal objectif des méthodes agiles est d'adapter les spécifications d'un projet suivant les retours d'informations de la part des utilisateurs. Des sondes décrivant le comportement des composants sont également utilisées pour pouvoir répondre à d'éventuels agissements erratiques peu voire pas visibles par les utilisateurs finaux. Ces méthodes ont ainsi réduit le fossé entre les équipes de développement et les utilisateurs finaux, et ont imposé des mécaniques telles que la livraison continue, c'est-à-dire la diffusion d'un nouveau livrable dès qu'une modification a été faite au code source et a été validée par les tests.

En parallèle, l'adoption de la méthodologie DevOps a permis de rapprocher les équipes de développement des équipes opérationnelles. Elle a notamment imposé la mise en œuvre de l'intégration continue, c'est-à-dire la mise en place automatisée dans une application du dernier livrable diffusé par l'équipe de développement. Pour traiter cette fonctionnalité, les applications

sont devenues de plus en plus dynamiques. Elles doivent en effet être capables de gérer l'ajout, la mise à jour et le retrait de leurs différents composants, en fonction de l'évolution de leur développement, et ce sans avoir à être redémarrée, afin de ne pas détériorer leur qualité de service. Lorsqu'une nouvelle version d'un composant est mise en place automatiquement (*rollforward*), elle est susceptible d'entraîner une baisse de la qualité de l'application globale sous de multiples formes. L'équipe opérationnelle peut, par exemple, être confrontée à des baisses de performance, une consommation excessive de ressources, etc.. De même, les utilisateurs peuvent subir l'apparition de bogues ou encore ne pas apprécier les modifications apportées par cette nouvelle version, *e.g.* de nouvelles interfaces homme-machine. Il doit alors être possible pour l'équipe opérationnelle de revenir rapidement à la version antérieure des composants modifiés (*rollback*). L'étape de *rollforward* est automatisable à l'aide de scripts ou réalisable directement par un opérateur, selon les outils dont dispose l'application mise à jour. La décision d'effectuer un *rollback* reste cependant à la discrétion de l'équipe opérationnelle et nécessite donc une intervention humaine pour être mise en œuvre.

Il apparait alors nécessaire de fournir aux opérateurs une automatisation de la gestion et de la surveillance de nouvelles versions de composants, ainsi que de l'annulation de leur déploiement.

8.1.2 Approche et exigences

L'approche que nous avons adoptée pour répondre à cette problématique a été de proposer la plate-forme Cohorte. Celle-ci permet la modélisation, l'exécution et la surveillance d'applications basées sur des composants.

La conservation d'une qualité de service acceptable malgré l'utilisation de composants potentiellement instables est assurée par un mécanisme d'isolation. Le cœur de l'application est protégé des erreurs des autres composants car il est exécuté dans un processus spécifique, voire sur une autre machine (physique ou virtuelle). Ces autres composants sont eux-mêmes regroupés dans différents isolats, soit automatiquement en fonction de leur langage d'implémentation et de leur fiabilité, soit selon une configuration préparée par l'équipe opérationnelle. Lorsque l'un de ces isolats s'arrête inopinément, *e.g.* suite à une erreur d'un composant, la plate-forme se charge de redistribuer les composants qu'il hébergeait et de les instancier à nouveau. L'application devient ainsi de plus en plus stable, les composants au comportement erratique étant de plus en plus isolés du reste de la composition.

Les exigences que nous retenons de la problématique sont les suivantes :

— la plate-forme doit supporter la mise à jour de composants à l'exécution, *i.e.* sans redémarrer l'application dans son intégralité ;

— de la même manière, la plate-forme doit supporter le retour à une version antérieure d'un composant ;

— la plate-forme doit protéger l'application du comportement erratique d'un composant, afin de ne pas engendrer l'arrêt de toute l'application : celle-ci doit toujours pouvoir fournir ses fonctionnalités basiques ;

— l'équipe de développement doit pouvoir se focaliser sur le code fonctionnel des composants et non sur leur intégration ;

— enfin, l'équipe opérationnelle doit pouvoir se concentrer sur la description de l'application et non sur la gestion de ses erreurs.

8.1.3 Contributions

L'ensemble des contributions décrites dans ce manuscrit forme la plate-forme Cohorte. La plate-forme ainsi que chacun de ses sous-projets sont diffusés sous les termes de la licence Apache Software License 2.0 [1] (ASL 2.0). Leur code source est disponible sur GitHub :

iPOPO :

`https://github.com/tcalmant/ipopo`

Cohorte Herald :

`https://github.com/isandlatech/cohorte-herald`

Cohorte Runtime et Composer :

`https://github.com/isandlatech/cohorte-runtime`

iPOPO

La première contribution de cette thèse est le modèle à composants orientés service iPOPO, avec sa plate-forme d'exécution, présenté dans le chapitre 3. Ses objectifs sont de proposer en Python des fonctionnalités et des facilités équivalentes à celle fournies en Java par les plates-formes à services OSGi et le modèle à composants iPOJO. Elle permet ainsi aux développeurs de se focaliser sur le développement du code fonctionnel de leurs composants, c'est-à-dire de ne pas avoir à traiter directement les liaisons entre composants. L'approche à services permet notamment aux composants qui fournissent les mêmes fonctionnalités d'être interchangeables à l'exécution, ainsi que de contrôler l'enregistrement de leurs services. Enfin, le choix du langage Python s'est imposé lors du développement de la surveillance des isolats, faisant appel à des méthodes de bas niveau exposées par des bibliothèques natives, *i.e.* développées en C.

Le projet iPOPO est composé de deux parties : Pelix et iPOPO. Pelix est la plate-forme d'architecture orientée service (SOA), qui implémente partiellement la spécification OSGi. Elle prend en charge le registre de services et les unités de déploiement (*bundles*). iPOPO est la plate-forme d'exécution du modèle à composants orientés service. Elle s'appuie sur les événements du registre de services de Pelix pour gérer les liaisons entre composants.

1. `http://www.apache.org/licenses/LICENSE-2.0`

Cette contribution a été adoptée pour développer le système de contrôle de la plate-forme Prédis MHI, HAL, présenté dans la section 7.3. Elle a également été utilisée lors de l'expérimentation présentée dans la section 7.4, mettant en œuvre le robot Nao comme interface homme-machine avec la plate-forme de contrôle domotique OpenHAB.

Cohorte Herald

La deuxième contribution de cette thèse est le canevas Cohorte Herald, qui permet aux composants d'une application Cohorte de communiquer. Il prend en charge l'émission et la réception de messages ainsi que le partage de services entre composants distants, tout en masquant le protocole de transport utilisé. Ce canevas étant générique et n'ayant pas besoin des concepts de Cohorte pour fonctionner, la dénomination *isolat* est remplacée par le terme *pair*. Un pair peut être défini comme étant une plate-forme d'exécution hébergeant les composants de Herald.

L'abstraction des protocoles de découverte et de transport sous-jacents est assurée par l'utilisation de services fournis par des extensions de Herald. Deux protocoles de transport sont actuellement supportés : HTTP et XMPP. Tandis que le service de transport XMPP est également capable de détecter les autres pairs utilisant ce protocole, du fait de l'utilisation d'un serveur central, il a été nécessaire de développer un service de découverte et son propre protocole pour les pairs s'appuyant sur HTTP. Ce protocole de découverte se base sur l'émission de paquets UDP *multicast* à intervalles réguliers, qui font office de *heart beat* (battement de cœur) indiquant et confirmant la présence de leur émetteur. Ces protocoles de découverte et de transport permettent aux composants d'une application répartie de communiquer par messages aussi bien localement, *via* HTTP, que depuis des réseaux distincts voire géographiquement éloignés, *via* XMPP. L'extensibilité de Herald autorise le développement d'autres services de découverte et de transport, ainsi que leur mise en œuvre immédiate dans chaque pair en cours d'exécution.

Herald fournit également une implémentation pour étendre Pelix Remote Services et sa version Java, Cohorte Remote Services, pour permettre aux composants de consommer les services enregistrés dans d'autres pairs, *i.e.* des services distants. Les mécanismes de découverte, d'import et d'export de services sont mis en place à travers des échanges de messages. Cette fonctionnalité est indispensable pour que les composants hébergés par différents pairs, *i.e.* dans différents isolats, puissent communiquer et donc pour qu'une application Cohorte fonctionne.

Cohorte Runtime

La troisième contribution présentée dans ce manuscrit est Cohorte Runtime. Elle prend en charge la gestion des isolats d'une application Cohorte, de leur démarrage à leur destruction, en passant par leur surveillance. Deux types d'isolats sont supportés : les isolats *exe*, *i.e.* des exécutables *boites noires* démarrés et surveillés par la plate-forme, et les isolats démarrés par *bootstrap* (amorce), dont le contenu est initialisé et maitrisé par Cohorte. Un isolat démarré par *bootstrap*,

appelé ci-après isolat, est un processus système hébergeant un ensemble d'artefacts (*bundles*) et de composants. Ceux-ci sont installés et instanciés par un script de démarrage (*bootstrap*). La plate-forme est contrôlée par un isolat *moniteur* et ne fonctionne que sur un seul nœud, c'est-à-dire une seule machine (physique ou virtuelle).

L'isolat *moniteur* est constitué d'une partie *Monitor*, chargée de la modélisation des isolats et de la réaction aux événements les concernant, et d'une partie appelée *Forker*, capable de démarrer les différents types d'isolats et de les surveiller suivant des mécanismes propres au système d'exploitation hôte. Le *Monitor* consomme le service fourni par le *Forker* pour démarrer les isolats qui auront été auparavant configurés par l'équipe opérationnelle, ainsi que pour redémarrer les isolats qui se seront arrêtés inopinément. Le *Forker* s'appuie sur des services *starters* pour démarrer les isolats en fonction de leur type (*exe*, *bootstrap*, ...) et sur un composant *Watcher* pour les surveiller. Il notifie le *Monitor* lorsqu'un isolat est considéré arrêté (panne franche, gel, ...).

Cohorte Runtime est le moteur de la plate-forme Cohorte. Sa capacité à démarrer et surveiller des isolats est indispensable au fonctionnement d'une application gérée par cette plate-forme. Ce canevas permet également à l'équipe opérationnelle de définir des isolats spécifiques, *i.e.* contenant des artefacts qui ne seront pas traités par Cohorte Composer, voire de faire démarrer et surveiller des outils tiers, tels qu'un serveur LDAP, comme des isolats.

Cohorte Composer

La dernière contribution de cette thèse est Cohorte Composer, qui permet de décrire une application comme un ensemble de composants. Ces composants sont définis dans des compositions et sont distribués à travers différentes machines (physiques ou virtuelles), appelées nœuds, et différents isolats. Chacun des noeuds sur lesquels doivent être instanciés des composants doivent exécuter une instance de Cohorte Runtime ; celle-ci étant utilisée pour gérer les isolats hébergeant ces composants.

Cohorte Composer est un assemblage de trois niveaux. Le *Top Composer* a pour tâche d'analyser une composition et de répartir ses composants sur différents nœuds, selon leur configuration. Il est généralement exécuté dans l'isolat *moniteur* d'un nœud, mais peut également être exécuté à part et être déconnecté de l'application une fois que ses ordres d'instanciation ont été transmis aux *Node Composers*. Le *Node Composer* est présent dans l'isolat *moniteur* de chaque nœud pouvant héberger des composants. Il calcule la distribution de ces composants entre différents isolats, en fonction des indications de la composition et de critères de fiabilité. Il demande ensuite au *Monitor* de Cohorte Runtime de démarrer ces isolats, si nécessaire. Chacun de ces isolats héberge un *Isolate Composer* qui reçoit ses ordres du *Node Composer* de son nœud. L'*Isolate Composer* s'appuie quant à lui sur des agents pour instancier, en fonction de leur modèle, les composants que doit héberger son isolat.

Le canevas Cohorte Composer est la pièce maitresse de la plate-forme Cohorte. Il permet à

l'équipe de développement et à l'équipe opérationnelle de définir une application par composition. Il prend en charge le démarrage et le maintien en exécution des composants de cette application. Le *Node Composer* est la partie la plus importante de Cohorte Composer. Elle a notamment la tâche de calculer une nouvelle distribution des composants après qu'un isolat se soit arrêté, en vue d'améliorer la qualité de service de l'application. Enfin, l'utilisation d'agents par l'*Isolate Composer* assure son extensibilité, en lui permettant de gérer de nouveaux modèles à composants.

Enfin, la table 8.1 présente le nombre de lignes de code (LOC) nécessaires à l'implémentation des différentes contributions de cette thèse. Seules les lignes contenant des instructions sont comptées ; les commentaires, la documentation et les lignes vides sont omis. Le nombre de lignes de code a été mesuré à l'aide de l'utilitaire *cloc.pl*[2].

Contribution	LOC Python	LOC Java	Python+Java
Pelix/iPOPO	10 504	0	10 504
Cohorte Herald	3 751	5 321	9 072
Cohorte Runtime	6 441	4 899	11 340
Cohorte Composer	2 151	630	2 781
Total	**22 847**	**10 850**	**33 697**

TABLE 8.1 – Lignes de code des différentes contributions

8.2 Perspectives

8.2.1 Travaux en cours

Les travaux en cours lors de la rédaction de ce manuscrit concernent Cohorte Herald.

Tunnels TCP et UDP

La première évolution de cet outil est la gestion de tunnels TCP et UDP, à la manière d'un tunnel SSH de type *local*. Comme le montre la figure 8.1, un tunnel a un point d'entrée, *i.e.* un port d'écoute ouvert par un pair, et un point de sortie, vers lequel le pair intermédiaire enverra les paquets qu'il reçoit. Les paquets sont transmis entre ces deux points à l'aide de messages Herald. Un client, qu'il soit un composant de l'application ou un outil tiers, peut alors envoyer des paquets vers le point d'entrée du tunnel pour qu'ils soient reçus par le serveur cible. L'intérêt de ce développement est de permettre à l'opérateur d'une application de profiter des capacités de transport de Herald afin d'interagir avec des serveurs difficilement accessibles : par exemple, un tunnel peut lui donner accès à un serveur protégé derrière un pare-feu en passant par la liaison XMPP utilisée

2. `http://cloc.sourceforge.net/`

par les pairs Herald, ou encore un tunnel peut permettre la simplification de la configuration de certains composants d'une application Cohorte. En effet, si l'un d'entre eux nécessite de dialoguer directement avec un serveur, *i.e.* par paquets réseau et non par services, et si ce dernier peut migrer d'un nœud à l'autre, alors il serait nécessaire de notifier à chaque fois ce composant de la nouvelle adresse dudit serveur. De fait, si un tunnel est créé et mis à jour par un composant spécialisé voire par le *Top Composer* (s'il est responsable de la migration du serveur) alors le composant pourrait être configuré une fois pour toute en indiquant une adresse qui lui serait locale (*localhost*) : les paquets seraient alors automatiquement redirigés vers leur cible.

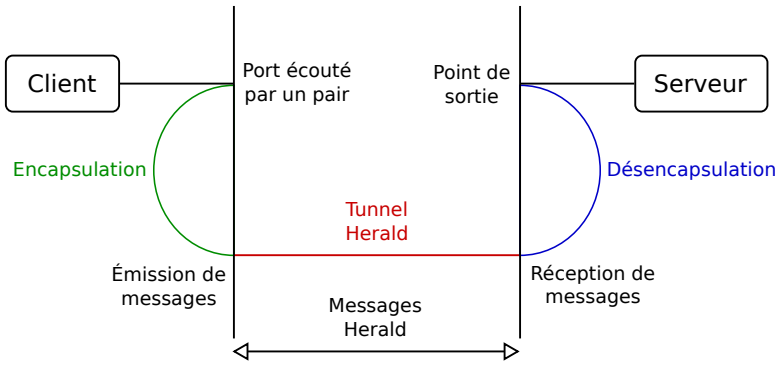

FIGURE 8.1 – Tunnel Herald

Routage des messages

Le second développement concernant Cohorte Herald apporte le routage de messages. Actuellement, il est nécessaire que deux pairs se soient découverts directement pour pouvoir dialoguer, ce qui signifie qu'ils doivent utiliser les mêmes mécanismes de découverte. De plus, leurs communications ne sont possibles que s'ils disposent d'au moins un service de transport en commun, *i.e.* s'ils savent utiliser les mêmes protocoles. Le routage de messages permettra aux pairs de se découvrir et de dialoguer à travers des pairs intermédiaires, appelés routeurs. À l'heure de la rédaction de ce manuscrit, les mécanismes d'interception par le système de routage ont été mis en place : lorsqu'un pair reçoit un paquet, il vérifie s'il en est le destinataire. Si c'est le cas, il traite le message, sinon il appelle son service de routage pour le transférer à sa cible ou au prochain pair intermédiaire. Le protocole de routage n'a pas encore été sélectionné, mais fera partie de la famille des protocoles à vecteurs de distances (*distance vector*). Cela signifie que chaque pair routeur connaitra tous les pairs de l'application et associera à chacun d'entre eux le prochain pair routeur à utiliser lors de la transmission d'un message, comme le montre la figure 8.2. Dans cet exemple, le pair `A` émet un message vers le pair `B`. Le routeur `R1`, auquel `A` est associé, dispose de deux routes pour accéder à `B`, en passant soit par `R2` soit par `R3`. Lorsque `R2` et `R3` ont notifié `R1` de leur connaissance de `B`, ils ont indiqué le nombre de sauts supplémentaires nécessaires pour

atteindre ce pair. `R1` a ainsi choisi `R2` pour transmettre ses messages à `B`, afin d'utiliser la route la plus courte. Si `R2` disparait ou perd son accès à `B`, `R1` pourra toujours utiliser la route secondaire *via* `R3`. Dans ce schéma, l'unité de distance est le nombre de sauts nécessaires pour atteindre le pair cible. La version qui sera implémentée s'appuiera sur le temps de transmission entre routeurs, afin d'emprunter le chemin le plus rapide plutôt que le plus court.

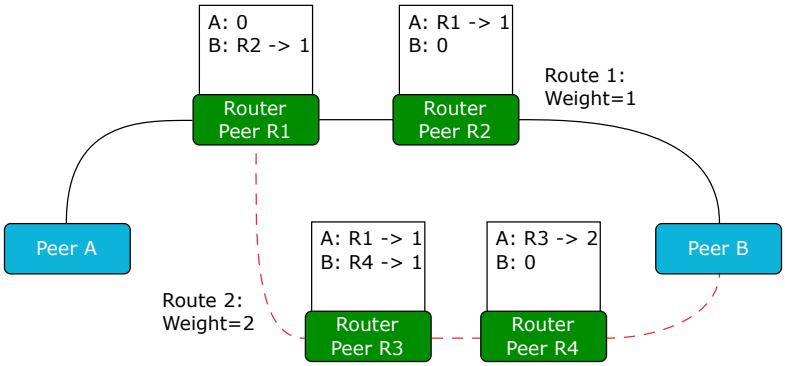

FIGURE 8.2 – Routage de messages Herald

Le routage de message sera notamment utilisé dans le cadre des services distants, du shell distant Herald et du mécanisme de tunnel présenté plus haut. Pour chacune de ces fonctionnalités, le routage permettra d'atteindre des pairs plus profondément enfouis dans l'architecture réseau. Ce développement fait partie du stage de magistère de Luc LIBRALESSO et fera l'objet d'une publication lorsqu'il aura été finalisé.

8.2.2 Travaux envisagés

Les travaux d'apport de fonctionnalités concernent principalement Cohorte Composer.

Support de nouveaux modèles à composants

La première étape consistera à développer des agents de l'*Isolate Composer* pour supporter plus de modèles à composants. Nous considérons importante la prise en charge de Declarative Services et Blueprint en Java, modèles spécifiés par OSGi, et Zope Component Architecture (ZCA) en Python, un des modèles à composants les plus utilisés dans ce langage. Ces développements auront pour principal objectif de rendre Cohorte Composer compatible avec des modèles populaires et donc de permettre la réutilisation d'un plus grand nombre de composants existants.

Critère de fiabilité du *Node Composer*

Afin d'être capable de gérer automatiquement le retour à une version stable d'un composant (*rollback*), les critères du *Node Composer* devront évoluer pour associer chaque note de fiabilité à

une version particulière du composant, et non au composant en général. De cette manière, ils seront en mesure de détecter une baisse de qualité de l'application due à la mise à jour du composant et de demander au *Node Composer* de réutiliser sa version précédente.

Description des nœuds pour le *Top Composer*

La description des *Node Composers* sera complétée afin de refléter les caractéristiques de son système hôte, *e.g.* le type de système d'exploitation (Posix, NT, ...), les versions de Java et Python disponibles, etc.. À l'aide de ces informations, le *Top Composer* pourra sélectionner les nœuds les plus appropriés pour héberger chaque composant, ou encore choisir quelle version de composant faire instancier sur chacun de ces nœuds. Le modèle de Cohorte Composer serait alors étendu afin de permettre d'indiquer pour chaque composant un filtre sur les capacités de son nœud hôte, plutôt que de ne donner que son nom.

Provisionnement

Une autre évolution du *Top Composer* sera mise en place afin de le lier à des outils de gestion d'infrastructure à grande échelle, *e.g.* OpenStack, voire à des orchestrateurs cloud, *e.g.* Roboconf, afin de démarrer automatiquement les nœuds nécessaires à l'hébergement des composants. De cette façon, l'opérateur d'une application Cohorte n'aura qu'à exécuter le *Top Composer* pour que l'intégralité de l'application, y compris son infrastructure, soit démarrée automatiquement. Un système de provisionnement, tel qu'Apache ACE [3] ou une implémentation de l'OSGi Bundle Repository (OBR, spécifié dans [OSG12b]), pourrait être mis en place afin de ne démarrer que des nœuds stéréotypés, *i.e.* ne contenant que la plate-forme Cohorte et un agent/client du système de provisionnement. Ces nœuds recevraient alors les artefacts nécessaires à l'instanciatation des composants qui leurs sont associés, sur commande du *Top Composer*. L'intérêt de ce mécanisme est double : (*i*) la gestion des modèles de nœuds est simplifiée car ceux-ci ne dépendent plus de l'application exécutée et (*ii*) la mise à jour des artefacts de l'application peut être effectuée à travers le système de provisionnement, plutôt que manuellement.

Intégration du canevas Cube

Afin d'avoir un comportement plus efficace de Cohorte Composer, notamment en ce qui concerne le passage à l'échelle, il est envisageable d'en développer une nouvelle implémentation en s'appuyant sur le canevas Cube [Deb14] de Bassem DEBBABI. Cube fournit une approche de composition basée sur des objectifs. Dans une composition, la description des composants serait ainsi remplacée par un archétype Cube, c'est-à-dire un graphe d'objectifs d'administration. Les

3. `https://ace.apache.org/`

composants sont les nœuds de ce graphe, tandis que leurs liaisons forment ses arêtes. La description des composants précise des contraintes sur leur contexte d'exécution, *i.e.* sur les propriétés des nœuds pouvant les héberger, ainsi que les propriétés initiales du composant. Les liaisons, *i.e.* les objectifs d'administration, associent quant à elles la description du composant consommateur à la description du composant fournisseur. À l'exécution, ces composants pourront être instanciés plusieurs fois, sur chaque nœud portant les propriétés de contexte nécessaires. Cette représentation plus complète permet une approche de liaison *a posteriori* (aussi appelée *lazy*, *i.e. fainéante*) des composants. Ainsi, étant donné que le *Composer* a connaissance des liens qu'il doit mettre en place, il peut optimiser la recherche des dépendances en se limitant à n'importer que les services nécessaires. À titre de comparaison, l'implémentation actuelle du *Composer* laisse la gestion des liaisons aux plates-formes d'exécution des modèles à composants. Pour garantir le bon fonctionnement de ces dernières, il est nécessaire d'importer tous les services exportés par les autres isolats. Cette approche est généralement appelée *greedy* (*gloutonne*). Elle est très consommatrice en ressources et en temps, du fait que chaque isolat connaisse tous les services de l'application.

Validation formelle de la plate-forme

Enfin, une autre phase de travail est prévue pour valider formellement les différents algorithmes de la plate-forme Cohorte. Jusqu'ici, leur fonctionnement tel que décrit dans ce manuscrit n'a été validé que par des tests unitaires et d'intégration, et par leur utilisation dans la plate-forme Agilium NG et ses propres tests.

BIBLIOGRAPHIE

[ACD+14] Shadi ABRAS, Thomas CALMANT, Benoît DELINCHANT, Stéphane PLOIX, Didier DONSEZ et Frédéric WURTZ : Power Management of Laptops Batteries in Dynamic Heterogeneous Environments Using iPOPO. *In the International Building Performance Simulation Association*, Arras, France, mai 2014. Springer.

[ACKM04] Gustavo ALONSO, Fabio CASATI, Harumi KUNO et Vijay MACHIRAJU : Web Services. *In Web Services*, Data-Centric Systems and Applications, pages 123–149. Springer Berlin Heidelberg, 2004.

[ACP+14] Shadi ABRAS, Thomas CALMANT, Stéphane PLOIX, Didier DONSEZ, Frédéric WURTZ, Olivier GATTAZ et Benoît DELINCHANT : Developing Dynamic Heterogeneous Environments in Smart Building Using iPOPO. *In The 3rd International Conference on Smart Grids and green IT Systems*, Barcelona, Spain, avril 2014. Springer.

[Amé13] João Claudio AMÉRICO : *A Component Metamodel for Designing Flexible and Modular Real-Time Applications*. Thèse de doctorat, Université de Grenoble, novembre 2013.

[And10] David J. ANDERSON : *Kanban*. Blue Hole Press, 2010.

[Ars04] Ali ARSANJANI : Service-oriented modeling and architecture. *IBM developer works*, pages 1–15, novembre 2004.
`http://www.ibm.com/developerworks/library/ws-soa-design1/`.

[Att14] Koutheir ATTOUCHI : *Managing resource sharing conflicts in an open embedded software environment*. Thèse de doctorat, Université Pierre et Marie Curie - Paris VI, juillet 2014.

[Bar12] Jonathan BARDIN : *RoSe : un framework pour la conception et l'exécution d'applications distribuées dynamiques et hétérogènes*. Thèse de doctorat, Université Joseph-Fourier - Grenoble I, octobre 2012.

[BBVB+01] Kent BECK, Mike BEEDLE, Arie VAN BENNEKUM, Alistair COCKBURN, Ward CUNNINGHAM, Martin FOWLER, James GRENNING, Jim HIGHSMITH, Andrew HUNT, Ron JEFFRIES *et al.* : The Agile Manifesto.
`http://agilemanifesto.org/iso/fr/`, février 2001.

[BCL+06] Eric BRUNETON, Thierry COUPAYE, Matthieu LECLERCQ, Vivien QUÉMA et Jean-Bernard STEFANI : The FRACTAL component model and its support in Java. *Software : Practice and Experience*, 36(11-12):1257–1284, 2006.

[BDF+03] Paul BARHAM, Boris DRAGOVIC, Keir FRASER, Steven HAND, Tim HARRIS, Alex HO, Rolf NEUGEBAUER, Ian PRATT et Andrew WARFIELD : Xen and the Art of Virtualization. *In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[Bec00] Kent BECK : *Extreme Programming Explained : Embrace Change*. The XP series. Addison-Wesley, 2000.

[Bel05] Fabrice BELLARD : QEMU, a Fast and Portable Dynamic Translator. *In Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[BFS+02] Rajesh BALAN, Jason FLINN, Mahadev SATYANARAYANAN, Shafeeq SINNAMOHIDEEN et Hen-I YANG : The Case for Cyber Foraging. *In Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 87–92, New York, NY, USA, 2002. ACM.

[Bin01] Walter BINDER : J-SEAL2 - A Secure High-Performance Mobile Agent System. *Electronic Commerce Research - Special issue on agents in electronic commerce*, 1(1-2):131–148, février 2001.

[BJPW99] Antoine BEUGNARD, Jean-Marc JEZEQUEL, Noël PLOUZEAU et Damien WATKINS : Making Components Contract Aware. *Computer*, 32(7):38–45, juillet 1999.

[BN84] Andrew D. BIRRELL et Bruce Jay NELSON : Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, février 1984.

[BOB92] Michael BLAIR, Sally OBENSKI et Paula BRIDICKAS : GAO/IMTEC-92-26 Patriot Missile Defense : Software Problem Led to System Failure at Dhahran, Saudi Arabia. Rapport technique, United States General Accounting Office, février 1992. `http://www.gao.gov/products/IMTEC-92-26`.

[Bou08] Johann BOURCIER : *Auto-Home : une plate-forme pour la gestion automatique d'applications pervasives*. Thèse de doctorat, Université Joseph-Fourier - Grenoble I, novembre 2008.

[CD01] Grzegorz CZAJKOWSKI et Laurent DAYNÉS : Multitasking Without Comprimise : A Virtual Machine Evolution. *In Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 125–138, New York, NY, USA, 2001. ACM.

[Cer04] Humberto CERVANTES : *Toward a service-oriented component model to support dynamic availability*. Thèse de doctorat, Université Joseph-Fourier - Grenoble I, mars 2004.

[CFH+05] Christopher CLARK, Keir FRASER, Steven HAND, Jacob Gorm HANSEN, Eric JUL, Christian LIMPACH, Ian PRATT et Andrew WARFIELD : Live Migration of Virtual Machines. *In Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[CHS+05] Grzegorz CZAJKOWSKI, Stephen HAHN, Glenn SKINNER, Pete SOPER et Ciarán BRYCE : A resource management interface for the Java™ platform. *Software : Practice and Experience*, 35(2):123–157, février 2005.

[CK13a] Stuart CHESHIRE et Marc KROCHMAL : DNS-Based Service Discovery. RFC 6763 (Proposed Standard). `https://tools.ietf.org/html/rfc6763`, février 2013.

[CK13b] Stuart CHESHIRE et Marc KROCHMAL : Multicast DNS. RFC 6762 (Proposed Standard). `https://tools.ietf.org/html/rfc6762`, février 2013.

[CSVC11] Ivica CRNKOVIĆ, Séverine SENTILLES, Aneta VULGARAKIS et Michel R.V. CHAUDRON : A Classification Framework for Software Component Models. *Software Engineering, IEEE Transactions on*, 37(5):593–615, septembre 2011.

[CvE98] Grzegorz CZAJKOWSKI et Thorsten von EICKEN : JRes : A Resource Accounting Interface for Java. *In Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 21–35, New York, NY, USA, 1998. ACM.

[Deb14] Bassem DEBBABI : *Cube : a decentralised architecture-based framework for software self-management*. Thèse de doctorat, Université de Grenoble, janvier 2014.

[DELD06] Mikael DESERTOT, Clement ESCOFFIER, Philippe LALANDA et Didier DONSEZ : Autonomic Management of Edge Servers. *In* Hermann de MEER et James P.G. STERBENZ, éditeurs : *Self-Organizing Systems*, volume 4124 de *Lecture Notes in Computer Science*, pages 216–229. Springer Berlin Heidelberg, 2006.

[DEM02] Frédéric DUCLOS, Jacky ESTUBLIER et Philippe MORAT : Describing and Using Non Functional Aspects in Component Based Applications. *In Proceedings of the 1st International Conference on Aspect-oriented Software Development*, AOSD '02, pages 65–75, New York, NY, USA, 2002. ACM.

[Dev15] Boris DEVOUGE : DevOps on Azure. `http://fr.slideshare.net/dataloop/dev-ops-on-azure`, 2015.

[DGR09] Didier DONSEZ, Kiev GAMA et Walter RUDAMETKIN : Developing Adaptable Components Using Dynamic Languages. *In Software Engineering and Advanced Applications, 2009. SEAA '09. 35th Euromicro Conference on*, pages 396–403, août 2009.

[DM09] Dan DRISCOLL et Antoine MENSCH : Devices Profile for Web Services. `http://docs.oasis-open.org/ws-dd/dpws/1.1/os/wsdd-dpws-1.1-spec-os.pdf`, juillet 2009.

[dSdRC14] Saulo Eduardo Galilleo Souza dos SANTOS, Tarcísio da ROCHA et Felipe Oliveira CARVALHO : HighFrame : An Integrated Solution for Developing Component-based Distributed Systems. *In Proceedings of the 7th Euro American Conference on Telematics and Information Systems*, EATIS '14, pages 15 :1–15 :6, New York, NY, USA, 2014. ACM.

[DT06] Didier DONSEZ et Gaël THOMAS : Propagation d'événements entre passerelles OSGi. *In Atelier de travail OSGi 2006*, Paris, France, 2006.

[EDH06] Clément ESCOFFIER, Didier DONSEZ et Richard S. HALL : Developing an OSGi-like service platform for .NET. *In Consumer Communications and Networking Conference, 2006. CCNC 2006. 3rd IEEE*, volume 1, pages 213–217, janvier 2006.

[ELR13] Clément ESCOFFIER, Philippe LALANDA et Nicolas REMPULSKI : H-ubu : An Industrial-strength Service-oriented Component Framework for JavaScript Applications. *In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 699–702, New York, NY, USA, 2013. ACM.

[ER15] Ahmed EL RHEDDANE : *Elasticity in the Cloud*. Thèse de doctorat, Université de Grenoble, février 2015.

[Esc08] Clément ESCOFFIER : *iPOJO : Un modèle à composant à service flexible pour les systèmes dynamiques*. Thèse de doctorat, Université Joseph-Fourier - Grenoble I, décembre 2008.

[Fed15] FEDERAL AVIATION ADMINISTRATION : Airworthiness Directives 2015-09-07. Rule 80 FR 24789, Federal Register, 2015. Docket No. FAA-2015-0936 Directorate Identifier 2015-NM-058-AD Amendment 39-18153 `https://federalregister.gov/a/2015-10066`.

[Fie00] Roy T. FIELDING : *Architectural Styles and the Design of Network-based Software Architectures*. Thèse de doctorat, University of California, 2000.

[GC03] Alan G. GANEK et Thomas A. CORBI : The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, 2003.

[GTM+09] Nicolas GEOFFRAY, Gaël THOMAS, Gilles MULLER, Pierre PARREND, Stéphane FRÉNOT et Bertil FOLLIOT : I-JVM : a Java Virtual Machine for Component Isolation

in OSGi. Research Report RR-6801, Institut National de Recheche en Informatique et en Automatique, 2009.

[HC01] George T. HEINEMAN et William T. COUNCILL : *Component-based Software Engineering : Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Hor01] Paul HORN : Autonomic computing : IBM's Perspective on the State of Information Technology.
`http://people.scs.carleton.ca/~soma/biosec/readings/autonomic_computing.pdf`, 2001.

[HT99] Andrew HUNT et David THOMAS : *The Pragmatic Programmer : From Journeyman to Master*. Addison-Wesley Professional, 1999.

[Jav00] JAVA COMMUNITY PROCESS : JSR 3 : Java™Management Extensions (JMX™) Specification.
`https://jcp.org/en/jsr/detail?id=3`, septembre 2000.

[Jav01] JAVA COMMUNITY PROCESS : JSR 19 : Enterprise JavaBeans™2.0.
`https://jcp.org/en/jsr/detail?id=19`, septembre 2001.

[Jav06] JAVA COMMUNITY PROCESS : JSR 121 : Application Isolation API Specification.
`https://jcp.org/en/jsr/detail?id=121`, juin 2006.

[Jav09] JAVA COMMUNITY PROCESS : JSR 284 : Resource Consumption Management API.
`https://jcp.org/en/jsr/detail?id=284`, janvier 2009.

[Jav11] JAVA COMMUNITY PROCESS : JSR 315 : Java™Servlet 3.0 Specification.
`https://jcp.org/en/jsr/detail?id=315`, mars 2011.

[JSO13] JSON-RPC WORKING GROUP : JSON-RPC 2.0 Specification.
`http://www.jsonrpc.org/specification`, avril 2013.

[KC03] Jeffrey O. KEPHART et David M. CHESS : The Vision of Autonomic Computing. *Computer*, 36(1):41–50, janvier 2003.

[KDSH11] Steffen KÄCHELE, Jörg DOMASCHKA, Holger SCHMIDT et Franz J. HAUCK : nOSGi : A Posix-compliant Native OSGi Framework. *In Proceedings of the 5th International Conference on Communication System Software and Middleware*, COMSWARE '11, pages 4 :1–4 :2, New York, NY, USA, 2011. ACM.

[KKL+07] Avi KIVITY, Yaniv KAMAY, Dor LAOR, Uri LUBLIN et Anthony LIGUORI : kvm : the Linux virtual machine monitor. *In Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[KOS+07] Kiyokuni KAWACHIYA, Kazunori OGATA, Daniel SILVA, Tamiya ONODERA, Hideaki KOMATSU et Toshio NAKATANI : Cloneable JVM : A New Approach to Start Isolated

Java Applications Faster. *In Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 1–11, New York, NY, USA, 2007. ACM.

[KW00] Poul-Henning KAMP et Robert N. M. WATSON : Jails : Confining the omnipotent root. *In Proceedings of the 2nd International SANE Conference*, 2000.

[Lag05] Menno LAGEMAN : Solaris Containers – What They Are and How to Use Them. Sun BluePrints™ Online, mai 2005.

[Lou11] Maxime LOUVEL : *Resources management and control architecture for multimedia applications in home networks*. Thèse de doctorat, Université de Bretagne occidentale - Brest, novembre 2011.

[LSG02] Nicolas LE SOMMER et Frédéric GUIDEC : JAMUS : Java Accommodation of Mobile Untrusted Software. *In NordU'02*, pages 38–48, Helsinki, Finland, février 2002. Multiprint, Helsinki.

[MRS11] Philippe MERLE, Romain ROUVOY et Lionel SEINTURIER : FraSCAti : Adaptive and Reflective Middleware of Middleware. *In 12th ACM/IFIP/USENIX International Middleware Conference*, Lisbonne, Portugal, décembre 2011.

[Ōno88] Taiichi ŌNO : *Toyota Production System : Beyond Large-Scale Production*. Productivity Press. Taylor & Francis, 1988.

[OSG05a] OSGI ALLIANCE : OSGi Service Platform Compendium Specification, Release 4. `http://www.osgi.org/Specifications`, août 2005.

[OSG05b] OSGI ALLIANCE : OSGi Service Platform Core Specification, Release 4. `http://www.osgi.org/Specifications`, août 2005.

[OSG12a] OSGI ALLIANCE : OSGi Service Platform Core Specification, Release 5. `http://www.osgi.org/Specifications`, mars 2012.

[OSG12b] OSGI ALLIANCE : OSGi Service Platform Enterprise Specification, Release 5. `http://www.osgi.org/Specifications`, mars 2012.

[OSG13] OSGI ALLIANCE : OSGi Service Platform Compendium Specification, Release 5. `http://www.osgi.org/Specifications`, mai 2013.

[OSG15] OSGI ALLIANCE : OSGi Service Platform Enterprise Specification, Release 6 (Early Draft 2). `https://osgi.org/download/osgi.enterprise-6.0.0-earlydraft2.pdf`, 2015.

[PTD+15] Linh-Manh PHAM, Alain TCHANA, Didier DONSEZ, Vincent ZURCZAK, Pierre-Yves GIBELLO et Noel DE PALMA : An adaptable framework to deploy complex applications onto multi-cloud platforms. *In Computing Communication Technologies - Research, Innovation, and Vision for the Future (RIVF), 2015 IEEE RIVF International Conference on*, pages 169–174, janvier 2015.

[PTS+11] Nicolas PALIX, Gaël THOMAS, Suman SAHA, Christophe CALVÈS, Julia LAWALL et Gilles MULLER : Faults in Linux : Ten Years Later. *In Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 305–318, New York, NY, USA, 2011. ACM.

[PvdH03] Mike P. PAPAZOGLOU et Willem-Jan van den HEUVEL : Service-oriented computing : concepts, characteristics and directions. *In Proceedings of the Fourth International Conference on Web Information Systems Engineering, 2003. WISE 2003.*, pages 3–12, décembre 2003.

[RA07] Jan S. RELLERMEYER et Gustavo ALONSO : Concierge : A Service Platform for Resource-constrained Devices. *In Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 245–258, New York, NY, USA, 2007. ACM.

[RAR07] Jan S. RELLERMEYER, Gustavo ALONSO et Timothy ROSCOE : R-OSGi : Distributed Applications Through Software Modularization. *In Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, Middleware '07, pages 1–20, New York, NY, USA, 2007. Springer-Verlag New York, Inc.

[Rie11] Eric RIES : *The Lean Startup : How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. Crown Publishing Group, 2011.

[RM09] Romain ROUVOY et Philippe MERLE : Leveraging component-based software engineering with Fraclet. *Annals of telecommunications - Annales des télécommunications*, 64(1-2):65–79, 2009.

[Roc13] James ROCHE : Adopting DevOps Practices in Quality Assurance. *Commun. ACM*, 56(11):38–43, novembre 2013.

[RR08] Leonard RICHARDSON et Sam RUBY : *RESTful Web Services*. O'Reilly Media, 2008.

[SB08] Ken SCHWABER et Mike BEEDLE : *Agile Software Development with Scrum*. Series in agile software development. Pearson Education International, 2008.

[SBM+14] Jetendr SHAMDASANI, Andrew BRANSON, Richard MCCLATCHEY, Coralie BLANC, Florent MARTIN, Pierre BORNAND, Sandra MASSONNAT, Olivier GATTAZ et Patrick EMIN : CRISTAL-ISE - Provenance Applied in Industry. *In Proceedings of the 16th International Conference on Enterprise Information Systems*, pages 453–458, 2014.

[SCA11] SCA : Service Component Architecture Assembly Model Specification, septembre 2011.

[SdG11] Kiev Santos da GAMA : *Towards Dependable Dynamic Component-based Applications*. Thèse de doctorat, Université de Grenoble, octobre 2011.

[SGM02] Clemens SZYPERSKI, Dominik GRUNTZ et Stephan MURER : *Component Software : Beyond Object-Oriented Programming*. ACM Press Series. ACM Press, 2002.

[SMF+09] Lionel SEINTURIER, Philippe MERLE, Damien FOURNIER, Nicolas DOLET, Valerio SCHIAVONI et Jean-Bernard STEFANI : Reconfigurable SCA Applications with the FraSCAti Platform. *In Services Computing, 2009. SCC '09. IEEE International Conference on*, pages 268–275, septembre 2009.

[SN05] Jim SMITH et Ravi NAIR : *Virtual Machines : Versatile Platforms for Systems and Processes*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2005.

[Som11] Ian SOMMERVILLE : *Software Engineering*. International Computer Science Series. Pearson, 2011.

[SPDC06] Lionel SEINTURIER, Nicolas PESSEMIER, Laurence DUCHIEN et Thierry COUPAYE : A Component Model Engineered with Components and Aspects. *In* Ian GORTON, George T. HEINEMAN, Ivica CRNKOVIĆ, Heinz W. SCHMIDT, Judith A. STAFFORD, Clemens SZYPERSKI et Kurt WALLNAU, éditeurs : *Component-Based Software Engineering*, volume 4063 de *Lecture Notes in Computer Science*, pages 139–153. Springer Berlin Heidelberg, 2006.

[Tou10] Lionel TOUSEAU : *SLA-driven Binding Policy for Intermittent Services*. Thèse de doctorat, Université Joseph-Fourier - Grenoble I, mai 2010.

[UPn08] UPNP FORUM : UPnP Device Architecture 1.1. `http://upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.1.pdf`, octobre 2008.

[VSDTD12] Tim VERBELEN, Pieter SIMOENS, Filip DE TURCK et Bart DHOEDT : AIOLOS : Middleware for Improving Mobile Application Performance through Cyber Foraging. *Journal of Systems and Software*, 85(11):2629–2639, 2012.

[Wal99] Jim WALDO : The Jini Architecture for Network-centric Computing. *Commun. ACM*, 42(7):76–82, juillet 1999.

[YSD+09] Bennet YEE, David SEHR, Gregory DARDYK, J. Bradley CHEN, Robert MUTH, Tavis ORMANDY, Shiki OKASAKA, Neha NARULA et Nicholas FULLAGAR : Native Client : A Sandbox for Portable, Untrusted x86 Native Code. *In 30th IEEE Symposium on Security and Privacy*, pages 79–93, mai 2009.